| | |
|---|---|
| **Titre:** Title: | Treatment of Degeneracy in Linear and Quadratic Programming |
| **Auteur:** Author: | Mehdi Towhidi |
| **Date:** | 2013 |
| **Type:** | Mémoire ou thèse / Dissertation or Thesis |
| **Référence:** Citation: | Towhidi, M. (2013). Treatment of Degeneracy in Linear and Quadratic Programming [Thèse de doctorat, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/1112/ |



| | |
|---|---|
| **URL de PolyPublie:** PolyPublie URL: | https://publications.polymtl.ca/1112/ |
| **Directeurs de recherche:** Advisors: | Dominique Orban, & François Soumis |
| **Programme:** Program: | Mathématiques de l'ingénieur |

UNIVERSITÉ DE MONTRÉAL

# TREATMENT OF DEGENERACY IN LINEAR AND QUADRATIC PROGRAMMING

MEHDI TOWHIDI
DÉPARTEMENT DE MATHÉMATIQUES ET DE GÉNIE INDUSTRIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(MATHÉMATIQUES DE L'INGÉNIEUR)
AVRIL 2013

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée :

# TREATMENT OF DEGENERACY IN LINEAR AND QUADRATIC PROGRAMMING

présentée par : TOWHIDI Mehdi
en vue de l'obtention du diplôme de : Philosophiæ Doctor
a été dûment acceptée par le jury d'examen constitué de :

M. EL HALLAOUI Issmaïl, Ph.D., président
M. ORBAN Dominique, Doct. Sc., membre et directeur de recherche
M. SOUMIS François, Ph.D., membre et codirecteur de recherche
M. BASTIN Fabian, Doct., membre
M. BIRBIL Ilker, Ph.D., membre

*To Maryam*
*To Ahmad, Mehran, Nima, and Afsaneh*

## ACKNOWLEDGEMENTS

Over the past few years I have had the great opportunity to learn from and work with professor Dominique Orban. I would like to thank him sincerely for his exceptional support and patience. His influence on me goes beyond my academic life.

I would like to deeply thank professor François Soumis who accepted me for the PhD in the first place, and has since provided me with his remarkable insight.

I am extremely grateful to professor Koorush Ziarati for introducing me to the world of optimization and being the reason why I came to Montreal to pursue my dream.

I am eternally indebted to my caring and supportive family ; my encouraging mother, Mehran ; my thoughtful brother, Nima ; my affectionate sister, Afsaneh ; and my inspiration of all time and father, Ahmad. I cannot express strongly enough how blessed I am.

Last, but by no means least, I owe this to my lovely wife, Maryam, for being there for me through ups and downs. Her cheerfulness and liveliness is the energy behind everything I do.

# RÉSUMÉ

Dans cette thèse, nous considérons la résolution de problèmes d'optimisation quadratique dégénérés sur base de techniques initialement développées pour l'optimisation linéaire, capables de tirer avantage de la dégénérescence. Nous commençons par améliorer l'efficacité de la règle de pivotage *positive edge* en fournissant une implémentation basée sur le logiciel libre CLP. Nous proposons ensuite le logiciel de haut niveau CyLP permettant de définir et d'expérimenter facilement avec de nouvelles règles de pivotage pour la méthode du Simplexe. CyLP offre de plus des services de modélisation puissants réduisant l'effort nécessaire à la modélisation de problèmes linéaires, en variables entières et quadratiques. À l'aide de CyLP, nous appliquons la règle *positive edge* à la variante du Simplexe suggérée par Wolfe pour résoudre les problèmes quadratiques. Nous incorporons également *positive edge* dans une méthode de gradient réduit. Nos tests démontrent l'efficacité de *positive edge* sur les problèmes quadratiques pour lesquels le terme linéaire est dominant. Chaque méthode est capable de fournir des niveaux substantiels d'accélération sur un certain sous-ensemble de problèmes lorsqu'elle est équipée de *positive edge*. Nous suggérons des pistes de recherche pour la conception de nouvelles méthodes qui incorporent *positive edge* et accélèrent la résolution sur une plus large gamme de problèmes.

# ABSTRACT

We consider solving degenerate quadratic programs (QPs) by means of degeneracy-benefiting techniques designed for linear programs (LPs). Specifically, we use a Simplex pivot method, called positive edge, that is able to take advantage of degeneracy in LPs. First, we improve the efficiency of the positive edge method by providing an internal implementation of it using CLP—an open-source LP solver. In the next stage, we develop a software, called CyLP, which allows easy definition of, and experimentation with, Simplex pivot rules. In addition, CyLP has a powerful modeling facility that reduces the effort of modeling LPs, mixed-integer programs (MIPs), and QPs. Using CyLP, we apply the positive edge rule to Wolfe's method—a Simplex-like method for QPs. We also incorporate positive edge into a reduced-gradient method. Our experiments demonstrate the effectiveness of positive edge on QPs with relatively large linear terms. Each method is able to yield substantial improvements on a subset of test problems. We provide research leads to devise novel methods that incorporate the positive edge rule and are more generally applicable.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

## LIST OF ACRONYMS AND ABBREVIATIONS

| | |
|---|---|
| LP | Linear Program |
| QP | Quadratic Program |
| SQP | Sequential Quadratic Programming |
| EQP | Equality Quadratic Programming |
| IPS | Improved Primal Simplex |
| VRP | Vehicule Routing Problem |
| NLP | Nonlinear program |

# CHAPTER 1

# INTRODUCTION

A Quadratic Program (QP) is a mathematical programming problem in which the objective function is quadratic and the constraints are linear. The importance of QP is twofold; first, it occurs naturally in many real life problems, e.g., the linear least-squares problem. Second, it is used in the solution process of general non-linear problems, e.g., Sequential Quadratic Programming (SQP) [Powell, 1978]. That is why QP is a central research area in mathematical programming. Gould and Toint [2008] gathered more than 1000 QP-related references, from 1943 to 2001.

In convex QPs, the objective function is convex and quadratic. Convex QPs and linear programs (LPs) share several similarities. Both categories of problems can be solved in polynomial time by way of interior-point methods. Moreover, the simplex method for LP generalizes to convex QP [Wolfe, 1959]. Active-set methods, which are among those resembling simplex method, explore faces of the feasible region to attain a solution. Like the simplex method, those routines struggle when facing degenerate problems.

Recently, constraint aggregation techniques were proven to be efficient on degenerate LPs [Elhallaoui et al., 2010a]. Those techniques are inspired by the special structure of set-partitioning problems, and generalize to LPs.

In this research, we use a technique to solve degenerate LPs—the positive edge method—to solve degenerate QPs.

In the first article, in Chapter 3, we present an implementation of the positive edge rule that outperforms the original implementation [Raymond et al., 2010a]. In Chapter 4, in the second article, we introduce a new software, CyLP, that allows easy definition and experimentation with customized pivot rules. In Chapter 5, we investigate the use of the positive edge method to solve degenerate QPs. We present a more detailed outline of the thesis in the following section.

### 1.1 Thesis Outline

Although the original implementation of the positive edge pivot rule, relying heavily on CPLEX, is efficient on highly degenerate instances, it struggles with relatively less degenerate problems. This is caused by implementation restrictions in CPLEX, as it does not allow user-defined pivots. Therefore the original implementation of positive edge is performed indirectly by solving a partial LP and having two procedures to supply it with compatible and incompatible variables (defined in Chapter 3). This causes overhead for the algorithm and poor performance on the mentioned instances.

Our goal is to use the positive edge rule to solve degenerate QPs. We begin by providing an efficient implementation of positive edge, using a two-dimensional reduced cost computation criteria. The idea is to assess the compatibility of a variable at the same time that we price variables, whether we use Dantzig's pivot, or steepest edge or a variant of it. In other words, when we search for a variable with a small-enough reduced cost, we also look for compatible variables with negative reduced cost. In the end, we choose the entering variable preferably from among the compatible variables, or else resort to incompatible variables. This preference is controlled by a parameter.

In order to implement the above strategy, we need an LP solver that allows user-defined customized Simplex pivot rules. We choose CLP, part of COIN-OR, which is open-source and free. In addition, its object-oriented structure facilitates development, experimentation, and modifications. Numerical tests show that on problems with more than 25% degeneracy the positive edge rule always speeds up the primal Simplex of CLP, with an average speedup of 2.

On the other hand, to solve a QP with Wolfe's method, we must implement Wolfe's pivot rule. Since CLP is written in the C++ programming language, it requires a knowledge of low-level programming for a user who wishes to modify it. As a result, the need to implement multiple pivot rule inspired the second contribution of the thesis. In this part, we developed an application, called CyLP, built upon CLP, that, among other features, allows us to define pivot rules in a high-level programming environment: Python. This makes the development stage shorter, and experimenting with the rule easier. For example, we implemented the positive edge rule in C++ in 106 lines, while implementing the same method in Python required only 38. Although this approach causes a slight loss of performance relative to a direct implementation in CLP, we demonstrate that when we analyze pivot rules, we obtain similar conclusions about their effectiveness, whether they are implemented in Python or in C++ .

Another great feature of CyLP that helps us in the next step is its modeling facility. It promotes modeling based on matrix algebra. We explain the details in Chapter 4, and we demonstrate how it makes modeling easier when we model Wolfe's LPs in Chapter 5.

Finally, in the third stage, we apply positive edge on Wolfe's method and the reduced gradient method to solve degenerate QPs. Using CyLP, we develop a Wolfe pivot rule that incorporates positive edge. Numerical tests lead us to identify some of the problem structures that cause the method to struggle. However, the reduced gradient method equipped with positive edge produces significant speedups on QPs with relatively large linear terms.

# CHAPTER 2

# LITERATURE REVIEW

Separate origins of the subjects that we consider during this research demand distinct sections devoted to each topic. Consequently, this chapter is organized as follows. In §2.1, we consider quadratic programs and their solution methods, in particular, the active-set methods. Afterwards, in §2.2, we provide background on the postive edge method, describing the research path that led to its introduction, which has its roots in constraint aggregation in set-partitioning problems.

## 2.1 Quadratic Program

A quadratic program (QP) is a mathematical optimization problem of the form

$$\begin{aligned} \underset{x\in\mathbb{R}^n}{\text{minimize}} \quad & c^T x + \tfrac{1}{2} x^T G x \\ \text{s.t.} \quad & Ax = b \\ & x \geq 0, \end{aligned} \tag{QP}$$

where $c \in \mathbb{R}^n$, $G \in \mathbb{R}^{n\times n}$ and is symmetric, $A \in \mathbb{R}^{m\times n}$, $b \in \mathbb{R}^m$ and the inequality $x \geq 0$ is understood elementwise. Note that any QP with inequality constraints $Ax \geq b$ can easily be converted to this form.

A QP is convex if and only if $G$ is positive semi-definite. Convex QPs are considered easier to solve, in the sense that there exist polynomial algorithms to find a global minimizer [Vavasis, 1991]. On the other hand, non-convex QPs are NP-hard. In fact, even finding a local minimizer of a non-convex QP is NP-hard [Vavasis, 1990].

### 2.1.1 Optimality Conditions

To introduce the optimality conditions, first we form the Lagrangian of (QP)

$$L(x, y, z) = \tfrac{1}{2} x^T G x + c^T x - y^T (Ax - b) - z^T x, \tag{2.1}$$

where $y \in \mathbb{R}^m$ and $z \in \mathbb{R}^n_+$ are the Lagrange multipliers.

The necessary conditions for optimality are

$$\nabla_x L(x, y, z) = Gx + c - A^T y - z = 0 \tag{2.2a}$$

$$Ax = b \tag{2.2b}$$

$$x_i z_i = 0 \quad \text{for} \quad i = 1, 2, \ldots, n \tag{2.2c}$$

$$(x, z) \geq 0, \tag{2.2d}$$

which are the Karush-Kuhn-Tucker (KKT) conditions [Nocedal and Wright, 1999].

When (QP) is convex, (2.2) are necessary and sufficient conditions for optimality in the sense that any solution to (2.2) is a global minimizer.

### 2.1.2 Algorithms

In an active-set method [Nocedal and Wright, 1999], at each iteration, we estimate the *active set*—the set of indices of the bounds for which equality holds at a solution. This estimate is based on the information gathered during previous iterations. By forcing equality on these constraints, we end up with a QP with equality constraints only. We solve this problem and use the results to correct our estimate of an optimal active set. Thus, at each iteration of an active-set method we solve an equality-constrained QP. That is why equality-constrained QP is a key to solving (QP). So we first consider the case of a convex QP with equality constraints.

For equality constrained QPs, the optimality conditions (2.2) reduce to the linear system

$$\begin{bmatrix} G & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} x \\ -y \end{bmatrix} = \begin{bmatrix} -c \\ b \end{bmatrix}. \tag{2.3}$$

The matrix on the left-hand side of the equation is called the KKT matrix. Note that this matrix, when $m \geq 1$, is always indefinite. There are several methods to solve (2.3). One is to factorize the KKT matrix and solve (2.3) directly. The Cholesky factorization is not suitable because of indefiniteness. For this purpose, a symmetric indefinite factorization is prefered which results in a factorization of the form

$$P^T K P = L B L^T, \tag{2.4}$$

where $K$ is the KKT matrix, $P$ is a permutation matrix, L is a unit lower triangular matrix (all the elements on the main diagonal are 1) and $B$ is a block-diagonal matrix with blocks of size either $1 \times 1$ or $2 \times 2$. So (2.3) can be solved by performing the following steps:

1. Solve $Lq = P^T \begin{bmatrix} -c \\ b \end{bmatrix}$ to obtain $q$.
2. Solve $B\hat{q} = q$ to obtain $\hat{q}$.
3. Solve $L^T\bar{q} = \hat{q}$ to obtain $\bar{q}$.
4. Set $\begin{bmatrix} x \\ y \end{bmatrix} = P\bar{q}$.

It is clear that the linear systems we solve in the first three steps are relatively easy because of the structures of the coefficient matrices.

There are several other methods to solve (2.3). Among them are the Shur-complement method, null-space method and the conjugate gradient method [Nocedal and Wright, 1999]. Although they are important numerical methods in practice, they are not necessary to understand our description of active-set methods.

Solving this system provides us with new primal variables and estimates of the optimal Lagrange multipliers. These multipliers allow us to either conclude optimality or that our estimate of the active constraints must be revised. In the second case, we update our estimate and perform a new iteration.

Another method is Wolfe's method in which we use a modified Simplex method to solve (QP). We solve a linear program with (2.2a), (2.2b) and (2.2d) as constraints. For the objective function, we minimize the values of artificial variables added to the constraints to obtain an initial point. Solving this LP with Wolfe's method gives a solution to (QP). Note that, to satisfy (2.2c), we need to modify the Simplex method so that a variable can only enter the basis if its dual variable is out of the basis and vice-versa. We explain these methods in the next chapter.

As in linear programming, degeneracy occurs in QP, but in different situations. For example, when the unconstrained minimizer of a QP is also the constrained minimizer and there are active constraints. Another type of degeneracy occurs when at a point, equality holds for some linearly dependent constraints. This situation may slow the solution process or even cause infinite cycling. We illustrate these two situations in Figure 2.1 and give examples in the numerical results section.

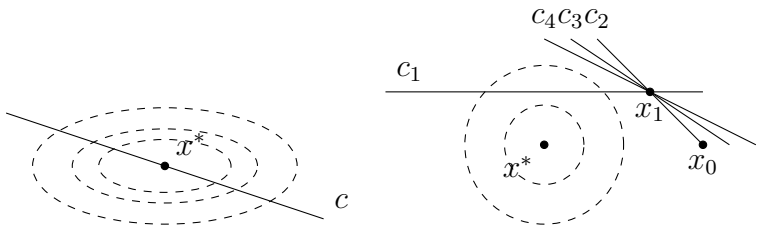



Figure 2.1 Different types of degeneracy in Quadratic Programming

### 2.1.3 Active-Set Methods

Active-set methods in quadratic programming are developed from both primal and dual perspectives. Before describing them, we briefly introduce the concepts and tools that are widely used by both of these methods. Later, we explain the primal and dual approaches and provide a comparison between them.

To solve (QP), one approach is to search on the faces of the feasible region which is very much like the Simplex method in linear programming. But note that in QP, optimality can occur anywhere alongside the faces or even inside the feasible region. Active-set methods perform such a search by using the active set. As stated earlier, the active set is our estimate of the bounds that are active at a solution. More precisely, for (QP), active set at $x$ is defined to be

$$\mathcal{A}(x) = \{i \mid x_i = 0\}, \tag{2.5}$$

However, to avoid redundancies in the bounds that we are keeping active and more importantly, to be able to loosen the active set to allow further displacements, we work with the *working set*, $\mathcal{W}(x)$—a subset of the active constraints at $x$, i.e. $\mathcal{W}(x) \subseteq \{i \mid x_i = 0\}$. In an active-set method, at each iteration $k$, given a working set $\mathcal{W}^k$ we solve the problem

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \quad & c^T x + \tfrac{1}{2} x^T G x \\ \text{s.t.} \quad & Ax = b \\ & x_i = 0 \quad \text{for} \quad i \in \mathcal{W}^k, \end{aligned} \tag{EP}$$

which we call the *equality problem*.

Primal and dual active-set methods use two operators in the course of their solution process. To explain the operators, we take a look at an approach to solve (2.3). Given that

the KKT matrix and $G$ are invertible, we can multiply both sides of (2.3) by

$$\begin{bmatrix} G & N^T \\ N & 0 \end{bmatrix}^{-1} = \begin{bmatrix} H & (N^*)^T \\ N^* & -\left(N^T G^{-1} N\right)^{-1} \end{bmatrix}, \tag{2.6}$$

where $N \in \mathbb{R}^{n \times |\mathcal{W}|}$ and the columns of $N$ are the coefficients of the constraints of (EP) and $H$ and $N^*$ are defined as follows:

$$N^* = (N^T G^{-1} N)^{-1} N^T G^{-1} \tag{2.7}$$

and

$$H = G^{-1}(I - NN^*) = G^{-1} - G^{-1}N(N^T G^{-1} N)^{-1} N^T G^{-1}. \tag{2.8}$$

Let $x_0$ and $x^*$ be a feasible point and the minimizer of (EP) respectively and $x^* = x_0 + s$. Also let $y_0$ be the vector of Lagrange multipliers corresponding to the active constraints at $x_0$. Using (2.6) to solve (2.3) for $x^*$ and $y^*$ we have

$$x^* = x_0 - Hg(x_0) \tag{2.9}$$

$$y^* = y_0 + N^* g(x_0)\,, \tag{2.10}$$

where $g(x_0)$ is the gradient of the objective function at $x_0$.

In effect, $H$ maps the gradient onto $x^* - x_0$ and $N^*$ maps the gradient onto $y^* - y_0$. Therefore $H$ and $N^*$ are used to perform primal and dual steps respectively. That is why many methods (like those that we explain in the next two sections) use these operators and the methods to update them when a modification is needed is a key to the performance of active-set methods.

### Primal Active-Set Method

In the primal active-set method at each iteration, we search for a solution to (EP) with respect to a specific $\mathcal{W}^k$, which is also feasible for (QP). The optimality conditions of (EP) reduce to

$$Gx^* + c - A^T y^* - E^k z^*_{\mathcal{W}^k} = 0 \tag{2.11a}$$

$$Ax^* = b \tag{2.11b}$$

$$x^*_i = 0 \quad \text{for} \quad i \in \mathcal{W}^k, \tag{2.11c}$$

where $x^*$ and $y^*$ are the primal solution and Lagrange multipliers with respect to the working set $\mathcal{W}^k$ and $E^k$ is in $\mathbb{R}^{n\times|\mathcal{W}^k|}$ all the rows of which are $e_i^T$ for $i \in \mathcal{W}^k$ ($e_i^T$ is the $i$-th row of the identity matrix) and finally $z^*_{\mathcal{W}^k}$ is in $\mathbb{R}^{|\mathcal{W}^k|}$ and contains only the variables $z_i$ where $i \in \mathcal{W}^k$. Moreover, $(x^*, y^*, z^*)$ is optimal for (QP) if it satisfies (2.11),

$$z^*_{\mathcal{W}^k} \geq 0, \text{ and} \tag{2.12}$$

$$x_i^* \geq 0 \quad \text{for} \quad i \notin \mathcal{W}^k, \tag{2.13}$$

where we set $z_i = 0$ for $i \notin \mathcal{W}^k$. Depending on whether or not $x^*$ is feasible for (QP) different situations may occur. We summarize a general primal active-set method in Algorithm 2.1.1.

**Algorithm 2.1.1** Outline of the Primal Active-Set Algorithm for QP

**Step 0.** Initialization: An initial feasible point, $x_0$, is given. Let $\mathcal{W}^0 = \mathcal{W}^k$ be a subset of active constraints at $x_0$.

**Step 1.** Obtain $x^*$ and $z^*$ satisfying (2.11).

**Step 2.** If $x^*$ is feasible to (QP):

a) If $z^* \geq 0$ stop ; $x^*$ is a solution for (QP).

b) Choose a constraint with a corresponding $z_i^* < 0$. Remove it from the working set and go to Step 1

**Step 3.** Choose a blocking constraint (a constraint that is not satisfied at $x^*$, i.e. for an $i$ we have $x_i^* < 0$) ; add it to the working set and go to Step 1.

Depending on the choice we make in Step 3 of Algorithm 2.1.1, in the presence of redundant constraints, we might end up cycling in a manner similar to that of linear programming if we do not use a safe guard, while adding and removing redundant constraints without moving in the feasible region. But if we manage to avoid this situation, we can prove that a working set will never be repeated in the process. Since the total number of possible working sets is finite, the algorithm terminates in a finite number of iterations [Nocedal and Wright, 1999].

Fletcher [1971] proposes a practical method to implement the primal active-set method by introducing a way to update the operators directly, after the working set is changed. The downside is that, as all primal active-set algorithms, it might suffer from degeneracy.

Goldfarb [1986] introduces another primal algorithm. The advantage of Goldfarb's primal algorithm is that it has the ability to remove multiple constraints at once if necessary, and a disadvantage is that we still need a feasible constrained minimizer as a starting estimate.

### Primal Active-Set Method and Simplex

It is informative to compare the Simplex method and the primal active-set method. For this section, consider the LP

$$\begin{aligned} \underset{x\in\mathbb{R}^n}{\text{minimize}} \quad & c^T x \\ \text{s.t.} \quad & Ax = b \\ & x \geq 0, \end{aligned} \tag{2.14}$$

where $A$, $b$ and $c$ are defined as in (QP).

In Table 2.1 we present the comparison where $x_B$, $x_N$, $x_{\mathcal{W}}$ and $x_{\mathcal{W}'}$ are the basic variables, non-basic variables, variables corresponding to the working set constraints and variables corresponding to $\mathcal{W}'$, respectively (where $\mathcal{W}'$ indexes the set of constraints not in the working set). Four different phases are selected for this comparison: starting phase, the phase in which a variable enters or leaves the basis and the optimality detection phase. We can verify the similarities between these two methods by comparing the results of each phase. On the other hand, there is a major difference. In the Simplex method, a variable enters the basis at the same time that another one leaves. But in the primal active-set method these occur independently, as you can see in Step 2.b and Step 3 in Algorithm 2.1.1. This allows an active-set method to search all the feasible points instead of just the extreme points of the feasible region.

### Dual Active-Set Method

The dual approach starts its search from the minimizer with respect to the equality constraints. Clearly, if QP has no equality constraints, the unconstrained minimizer $x_0 = -G^{-1}c$ is the starting point. If this point is feasible for (QP), it is the solution to the problem. Otherwise we start by adding constraints to the working set one by one and remove them when necessary with the goal of finally obtaining the correct estimate of the active set at an optimal solution. In the dual approach, at each iteration, the current point is the minimizer of the objective function of (QP), subject to the active constraints, but it is not necessarily feasible for (QP).

Table 2.1 Simplex vs. Primal Active-set Method

| **Simplex** | | **Primal Active-Set Method** | |
|---|---|---|---|
| Phase | Result | Phase | Result |
| Start from a basic feasible point. | $x_N = 0$, $x_B \geq 0$ | Start from a feasible point with respect to a working set $\mathcal{W}$. | $x_{\mathcal{W}} = 0$, $x_{\mathcal{W}'} \geq 0$ |
| Compute the reduced cost $r = c - A^T y$ and find a variable, $x_e$, with negative reduced cost to *enter* the basis. | $x_e \geq 0$ | Solve (2.3), compute $z = Gx + c - A^T y$ and find a constraint, $x_e = 0$, with negative $z$ to *remove* from the working set. | $x_e \geq 0$ |
| Perform the ratio test to determine the *leaving* variable from the basis, $x_l$. | $x_l = 0$ | Perform the ratio test to determine a blocking constraint $x_l \geq 0$ to *enter* the working set. | $x_l = 0$ |
| Detect optimality | $r \geq 0$ | Detect optimality | $z \geq 0$ |

The dual of (QP) is

$$\begin{aligned} \underset{x,y,z}{\text{maximize}} \quad & b^T y - \tfrac{1}{2} x^T G x \\ \text{s.t.} \quad & Gx + c = A^T y + z \\ & z \geq 0, \end{aligned} \tag{D}$$

where $x$, $y$ and $z$ are defined as before. Consequently, we call a triplet $(x, y, z)$ to be *dual feasible* if it satisfies both constraints of (D).

An outline of the dual active-set method is shown in Algorithm 2.1.2.

From (D) and Algorithms 2.1.1 and 2.1.2, we can interpret comprehensibly the steps taken in the course of the primal and dual active-set methods; we are looking for primal and dual feasibility. In other words, in the dual method, we keep (2.11) satisfied while searching for a feasible point for (QP). This is in contrast with the primal active-set method where we keep primal feasibility and (2.11c) holding and search for a point that is dual feasible (that satisfies (2.11a) and (2.13).

**Algorithm 2.1.2** Outline of the Dual Active-Set Algorithm for QP

**Step 0.** Initialization: let $x$ be an unconstrained minimizer of (QP).

**Step 1.** If $x$ is feasible for (QP), stop ; $x$ is optimal.

**Step 2.** Find an inequality constraint that is not satisfied. Use it (together with the operators (2.7) and (2.8)) to compute a direction in the primal and in the dual space.

a) If no point along the primal direction can satisfy the violated constraint:

1) If the dual direction never violates the dual-feasibility then stop ; (QP) is infeasible.

2) Take the largest possible step in the dual space, i.e., until a dual variable becomes zero. Remove the new inactive constraint from the working set (Note that this is a pure dual step with no displacement in the primal space). Go to Step 2.

b) Move in the primal and dual directions and stop whenever we are about to loose dual-feasibility or have satisfied the violated constraint, whichever comes first. In the former case remove the new inactive constraint from the working set and go to Step 2. In the latter, add the new active constraint to the working set and go to Step 1.

Goldfarb and Idnani [1983] present a dual active-set algorithm. It makes use of the same operators as (2.7) and (2.8) to obtain the primal and dual directions. One of the great advantages of the method is that it provides an efficient method to update the two operators indirectly when adding or removing a constraint. While proven to be highly efficient compared to Fletcher's primal method, it has deficiencies when we search for a minimizer that occurs at a vertex of the feasible region [Goldfarb and Idnani, 1983].

### Comparing Active-Set Methods

Figure 2.2 shows different approaches of the primal and dual active-set methods in a QP with inequality constraints only. The dashed circles show the level curves of a quadratic function. There are four inequality constraints $c_1$ to $c_4$ which describe the hatched triangular feasible region. For each constraint, the side of the label specifies the side of the constraint which is feasible. The primal method (the blue path) is considered to start at a given $x_0$ and an initial working set $\mathcal{W}_0 = \{3\}$. Note that the initial active set is $\mathcal{A}_0 = \{2, 3\}$. We could have obtained the solution in the first step had we chosen $\mathcal{W}_0 = \{2\}$ or $\mathcal{W}_0 = \varnothing$. It becomes clear how the choice of working set can affect the primal method. Another observation about the primal process is that although all the end-points of the path lay on a face of the feasible region, we may have a displacement that passes through the interior of the feasible region. On

the other hand, the dual approach (the red path) starts from the unconstrained minimizer of the QP, $x^u = -G^{-1}c$. We can see that the most violated constraint is not necessarily active at the solution. In the example $c_4$ is violated the most at $x_0$.

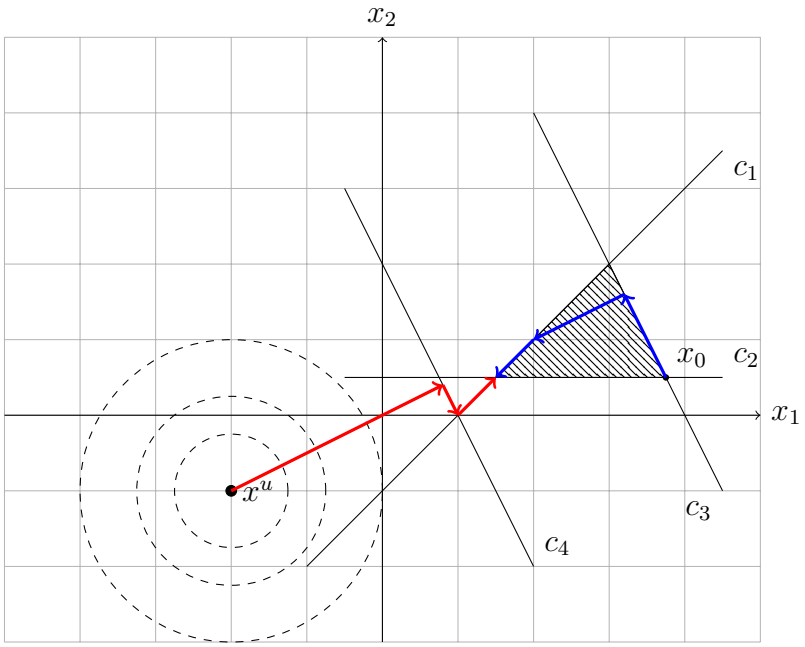



Figure 2.2 Active-set method, primal (the blue path) and dual (the red path)

### Numerical Examples

To illustrate the primal and dual active-set methods, we apply them to two degenerate QPs. In this section we consider QPs of the form

$$\begin{aligned} \underset{x\in\mathbb{R}^n}{\text{minimize}} \quad & c^T x + \tfrac{1}{2} x^T G x \\ \text{s.t.} \quad & Ax \geq b. \end{aligned}$$

As mentioned earlier, this formulation is equivalent to (QP).

**Example 1**

Consider the following QP

$$\begin{array}{lrcrcl}
\text{minimize} & \frac{1}{2}x_1^2 & + & \frac{1}{2}x_2^2 & & \\
\text{s.t.} & -\frac{2}{3}x_1 & - & x_2 & \geq & -\frac{7}{3} \\
& -x_1 & - & x_2 & \geq & -3 \\
& -\frac{1}{2}x_1 & - & x_2 & \geq & -2 \\
& & - & x_2 & \geq & -1 \quad .
\end{array}$$

So we have

$$G = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \qquad A = \begin{bmatrix} -\frac{2}{3} & -1 & -\frac{1}{2} & 0 \\ -1 & -1 & -1 & -1 \end{bmatrix}^T,$$

$$c = \begin{bmatrix} 0 & 0 \end{bmatrix}^T \quad \text{and} \quad b = \begin{bmatrix} -\frac{7}{3} & -3 & -2 & -1 \end{bmatrix}^T.$$

We solve the problem using the primal active-set method. Suppose that we have $x_0 = \begin{bmatrix} 3 & 0 \end{bmatrix}^T$ and $\mathcal{W}_0 = \{2\}$. Steps of the primal active-set method are shown in Table 2.2. In this table $d$, $t$ and $f$ denote the direction, the step length and the objective function value respectively ; $\mathcal{W}$ is the working set and $z$ is the vector of Lagrange multipliers.

Figure 2.3 shows the solution process. Dotted circles show the level curves of the objective function, the blue path shows the displacements and $c_1$ through $c_4$ are the four constraints of the problem in order.

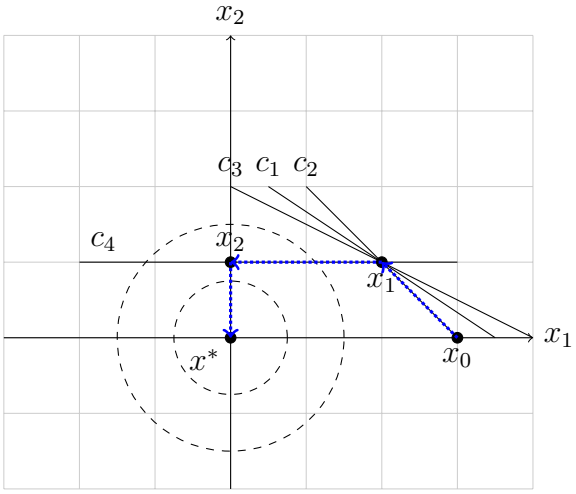



Figure 2.3 Primal active-set method, example 1

Note that during six iterations, there is no change in the objective function. In fact, we must perform two iterations before we notice that we added the "wrong" constraint to the working set. Thus, each wrong decision when facing blocking constraints causes to solve (2.11)

Table 2.2 Primal active-set method, example 1

| Iteration | $x$ | $f$ | $\mathcal{W}$ | $z$ | d | t | Blocking constraints |
|---|---|---|---|---|---|---|---|
| 1 | $\begin{bmatrix} 3 \\ 0 \end{bmatrix}$ | $\frac{9}{2}$ | $\{2\}$ | $\begin{bmatrix} -1.5 \end{bmatrix}$ | $\begin{bmatrix} -1.5 \\ 1.5 \end{bmatrix}$ | $\frac{2}{3}$ | $\{1,3,4\}$ |
| 2 | $\begin{bmatrix} 2 \\ 1 \end{bmatrix}$ | $\frac{5}{2}$ | $\{1,2\}$ | $\begin{bmatrix} 3 \\ -4 \end{bmatrix}$ | $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$ | - | - |
| 3 | $\begin{bmatrix} 2 \\ 1 \end{bmatrix}$ | $\frac{5}{2}$ | $\{1\}$ | $\begin{bmatrix} -1.61 \end{bmatrix}$ | $\begin{bmatrix} -0.92 \\ 0.61 \end{bmatrix}$ | 0 | $\{2,3,4\}$ |
| 4 | $\begin{bmatrix} 2 \\ 1 \end{bmatrix}$ | $\frac{5}{2}$ | $\{1,3\}$ | $\begin{bmatrix} -9 \\ 8 \end{bmatrix}$ | $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$ | - | - |
| 5 | $\begin{bmatrix} 2 \\ 1 \end{bmatrix}$ | $\frac{5}{2}$ | $\{3\}$ | $\begin{bmatrix} -1.6 \end{bmatrix}$ | $\begin{bmatrix} -1.2 \\ 0.6 \end{bmatrix}$ | 0 | $\{1,2,4\}$ |
| 6 | $\begin{bmatrix} 2 \\ 1 \end{bmatrix}$ | $\frac{5}{2}$ | $\{3,4\}$ | $\begin{bmatrix} -4 \\ 3 \end{bmatrix}$ | $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$ | - | - |
| 7 | $\begin{bmatrix} 2 \\ 1 \end{bmatrix}$ | $\frac{5}{2}$ | $\{4\}$ | $\begin{bmatrix} -1 \end{bmatrix}$ | $\begin{bmatrix} -2 \\ 0 \end{bmatrix}$ | 1 | - |
| 8 | $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$ | $\frac{1}{2}$ | $\{\}$ | $[]$ | $\begin{bmatrix} 0 \\ -1 \end{bmatrix}$ | 1 | - |
| 9 | $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$ | 0 | $\{\}$ | $[]$ | $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$ | - | - |

two times more than necessary. Had we chosen constraint $c_4$ to become active at iteration 2, we would have saved four iterations.

Note also the chance to cycle infinitely. Since the four constraints of the problem coincide at $\begin{bmatrix} 2 & 1 \end{bmatrix}^T$, it is possible to add and remove constraints $c_1$, $c_2$, and $c_3$ to and from the working set and ignore constraint $c_4$, which is the one that allows displacement.

We also point out that in this example, the dual active-set method finds the solution at the first iteration since the unconstrained minimizer is the solution.

**Example 2**

The problem is to

$$\begin{array}{lrcrcr} \text{minimize} & \frac{1}{2}x_1^2 & + & \frac{1}{2}x_2^2 & - & 2x_1 \\ \text{s.t.} & \frac{2}{3}x_1 & + & x_2 & \geq & \frac{7}{3} \\ & x_1 & + & x_2 & \geq & 3 \\ & \frac{1}{2}x_1 & + & x_2 & \geq & 2 \\ & & + & x_2 & \geq & 1 \quad . \end{array}$$

We have

$$G = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \qquad A = \begin{bmatrix} \frac{2}{3} & 1 & \frac{1}{2} & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}^T,$$

$$c = \begin{bmatrix} -2 & 0 \end{bmatrix}^T \text{ and } \qquad b = \begin{bmatrix} \frac{7}{3} & 3 & 2 & 1 \end{bmatrix}^T.$$

We solve this problem using the dual active-set method. The process is shown in Table 2.3. In this table $x$, $f$, $\mathcal{W}$ and $z$ are defined as in Example 1.

Table 2.3 Dual active-set method, example 2

| Iteration | $x$ | $f$ | $\mathcal{W}$ | $z$ | Primal step | Dual step | Blocking const. |
|---|---|---|---|---|---|---|---|
| 1 | $\begin{bmatrix} 2 \\ 0 \end{bmatrix}$ | -2 | {} | - | $\begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}$ | $\begin{bmatrix} 0.5 \end{bmatrix}$ | {2} |
| 2 | $\begin{bmatrix} 2.5 \\ 0.5 \end{bmatrix}$ | -1.75 | {2} | $\begin{bmatrix} 0.5 \end{bmatrix}$ | $\begin{bmatrix} -0.5 \\ 0.5 \end{bmatrix}$ | $\begin{bmatrix} -0.5 \\ 1 \end{bmatrix}$ | {1, 3, 4} |
| 3 | $\begin{bmatrix} 2 \\ 1 \end{bmatrix}$ | -1.5 | {2, 4} | $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$ | - | - | - |

Figure 2.4 shows the solution process. The blue path starts with the unconstrained minimizer, $x_0$, and ends at the solution, $x^*$.

At iteration 3, where three weakly active constraints exist, we stop the procedure because all the constraints are satisfied and so the point is feasible. This implies that in the dual active-set method we simply ignore any weakly active constraint not in the working set at any given point.

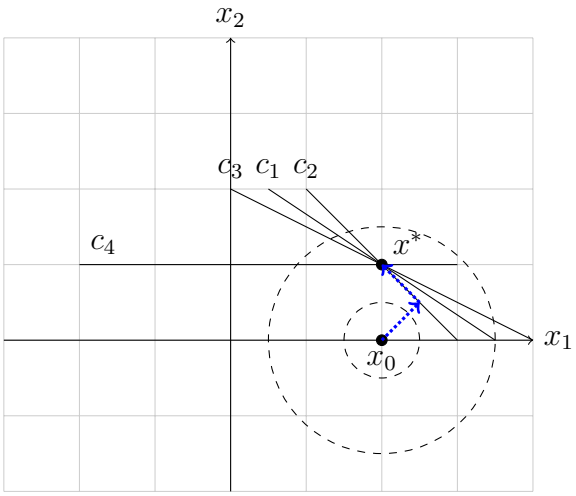



Figure 2.4 Dual active-set method, example 2

### Modifications to Active-Set Methods

Moré and Toraldo [1989] present a modification to the standard primal active-set method. It uses the conjugate gradient method to solve the KKT system arising at each iteration of an active-set approach for QP problems subject to bounds. It is shown that for non-degenerate problems, this method reduces the number of iterations by at least a factor of 10 compared to the standard primal method. On degenerate QP problems, depending on the degree of degeneracy, it performs less efficiently, but still better than the standard primal method.

As a modification to the mentioned method, Moré and Toraldo [1991] combine their method with a gradient projection method for the solution of large-scale QPs. While still searching for a solution with respect to the current working set using the conjugate gradient method, they move from one working set to another using the gradient projection method which allows adding multiple constraints to the working set in a single move. Since most

of the solution time is consumed by solving the linear system (2.3), being able to use that solution to add more than one constraint whenever possible is an advantage of this method.

### 2.1.4 Simplex Extensions for QP

To find a solution to (QP) and equivalently a solution to the optimality conditions (2.2), many Simplex-like methods have been proposed. Among the known methods are Wolfe's method [Wolfe, 1959], Dantzig's method [Dantzig, 1963] and Lemke's method [Fletcher, 1987]. Here we are more interested in the methods that have a close similarity to the Simplex method. Wolfe [1959] proposes to consider (2.2a), (2.2b) and (2.2d) as the constraints of an LP and take care of constraint (2.2c) in the solution process. The linear program, after adding artificial variables $a_1 \in \mathbb{R}^n$, $a_2 \in \mathbb{R}^m$, is

$$\begin{aligned} \underset{x,y,z}{\text{minimize}} \quad & e_1^T a_1 + e_2^T a_2 \\ \text{s.t.} \quad & Gx - A^T y - z + a_1 = -c \\ & Ax + a_2 = b \\ & (x, z, a_1, a_2) \geq 0 \,, \end{aligned} \tag{2.15}$$

where all the elements of $e_1$ (an $n$-vector) and $e_2$ (an $m$-vector) are equal to one. In (2.15), we are looking for a feasible solution rather than minimizing an objective function. In effect, the goal is to minimize the sum of the artificial variables. From (2.2c), between $x_i$ and $z_i$ at least one should be zero at a solution of (QP). For this reason, we call $x_i$ and $z_i$ to be the complement of each other. To satisfy the complementary conditions (2.2c), we slightly modify the selection rule for the variable entering the basis. We choose a variable with the most negative reduced cost to enter the basis with the condition that its complement is non-basic or will exit the basis at the same iteration.

When $G$ is positive definite, this method is guaranteed to converge in at most $\binom{3n}{n}$ iterations [Wolfe, 1959].

## 2.2 Constraint Aggregation

### 2.2.1 Set-Partitioning Problems

Set-partitioning problems are special cases of MIPs and appear in many real world problems, such as scheduling problems and vehicle routing problems (VRP). A set-partitioning problem is an optimization problem of the form

$$\begin{aligned} \underset{x\in\mathbb{R}^n}{\text{minimize}} \quad & c^T x \\ \text{s.t.} \quad & Ax = e \\ & x \in \mathcal{B}^n, \end{aligned} \tag{2.16}$$

where $\mathcal{B} = \{0, 1\}$, $c \in \mathbb{R}^n$, $A \in \mathcal{B}^{m\times n}$ and $e$ is an $m$-vector with all of its element equal to one. In a scheduling problem, each constraint corresponds to a *task* that should be performed exactly once and each variable represents a way to perform some tasks, e.g. a *route* or *path* in a VRP. A VRP can be represented by a network consisting of nodes corresponding to tasks that should be performed, and arcs between the nodes showing the possible displacements. There are vehicles that start from a depot, move along a chosen path and perform the tasks as they pass the nodes. A cost is associated with each arc and we try to minimize the cost. For example, consider that we want to service three customers residing in different parts of a city. We have two vehicles available. Suppose that a vehicle can only use one of the following paths: the first path that passes by customer 1, the second path that passes by customer 2, the third path that passes by customers 1 and 3 and finally, the fourth path that passes by customers 2 and 3. We define variable $x_i$ to be 1 if we choose the path $i$ and 0 otherwise. The cost of using path $i$ is $c_i$. The problem can be stated as the MIP

$$\begin{aligned} \text{minimize} \quad & c_1x_1 + c_2x_2 + c_3x_3 + c_4x_4 \\ \text{s.t.} \quad & x_1 + x_3 = 1 \\ & x_2 + x_4 = 1 \\ & x_3 + x_4 = 1 \\ & x_i \in \mathcal{B} \quad \text{for} \quad i = 1, 2, 3, 4\ . \end{aligned}$$

Note that each constraint corresponds to a task which should be performed exactly once and each variable corresponds to a path.

Methods to solve (2.16) (e.g. branch and bound methods) need to solve a relaxation of the problem at each iteration. This relaxation is obtained by replacing constraints $x \in \mathcal{B}^n$ in

(2.16) by $0 \leq x \leq 1$. The relaxation of (2.16) is an LP and we can use the Simplex method to solve it. However, in practice, set-partitioning problems are highly degenerate, i.e., at any given iteration, a high percentage of the variables in the Simplex basis are zero. Degeneracy can cause cycling in the absence of a safeguard or at least stalling in the solution process.

Different methods have been proposed to reduce the effect of degeneracy. Among them, using constraint aggregation for this purpose is relatively new. Elhallaoui et al. [2005] propose a method called Dynamic Constraint Aggregation (DCA). The idea of DCA is to remove the zero variables from the basis. Considering only the positive-valued variables in the basis, some constraints become identical. We keep one of the identical constraints in the problem and remove the others temporarily. This way, we work with a smaller basis and as long as we fix the basic zero variables to zero, we are sure that the removed constraints are satisfied.

Methods to be discussed in this section are modifications to the standard *column generation* (CG) method. CG solves a problem (here, for example, a VRP) by iteratively solving a problem called the master problem and one, or multiple, subproblems. The master problem is a linear programming problem which consists of a relatively small number of paths. Subproblem(s) provide the master problem with profitable paths—those with negative reduced costs. When no more paths with negative reduced cost exist, we have found the solution. Subproblems are usually of a type that are relatively easy to solve. For example in a VRP the subproblem is an elementary shortest path problem.

DCA is based on a definition for equivalence of tasks: two tasks are equivalent if for every path in a set of paths, $C$, they are both covered or none of them is covered (the set $C$ is initially generated from some seemingly good paths that are possibly infeasible). Using this definition, the tasks are partitioned into several equivalence classes, called clusters. Let $L$ be the set of these equivalence classes and for all $l \in L$, $W_l$ be the set of tasks belonging to cluster $l$. A *partition* of tasks is defined to be $Q = \{W_l \,|\, l \in L\}$. If a new variable (path) is about to enter the basis, it should be compatible with this partition in the sense that for each equivalence class, it should cover all the tasks in the class or none of them. The general idea is to keep only one constraint representing each cluster in the master problem and temporarily remove the others. If there are no more compatible variables with the current partition, and yet there are still negative reduced-cost variables, the partitioning is updated by breaking up some clusters and adding a subset of incompatible variables.

The process occurs in two nested loops called major and minor iterations. During a minor iteration, we fix the partition and try to find eligible variables, i.e. with negative reduced cost and compatible with the current partition, to enter the basis. If this is not possible, a major iteration is needed, in which we either conclude optimality or change the partition so that

some eligible but incompatible variables become compatible. The partition is updated when it is no longer possible to generate compatible variables , or the reduced cost of incompatible variables are considerably smaller than the compatible variables.

An important detail of the method is the need to recover the dual variables corresponding the temporarily ignored constraints, in order to compute the reduced costs. All that is known is the sum of the dual variables of the tasks in each cluster. The process of extracting the values of the dual variables is called dual disaggregation. This problem is reduced to a shortest path problem created using the dual constraints. This method aims at finding dual variables that create positive reduced costs for all the incompatible variables and proving optimality. Figure 2.5 presents a flowchart of the DCA algorithm.

Elhallaoui et al. [2005] report a speedup of 5 for DCA relative to the traditional column generation method. An average 39% decrease in the number of constraints and 90% decrease in the master problem solution time is also reported with specific problems.

One disadvantage of the DCA method is that it does not distinguish between different incompatible variables. It ignores that some incompatible variables with good reduced costs may cause the partition size to grow rapidly, while other incompatible variables with less favorable reduced costs may cause a less speedy grow. To address this issue, Elhallaoui et al. [2010b] propose a Multi-phase DCA. They define a scale for evaluating the level of incompatibility of a variable with the current partition. Then the problem is solved in different phases, such that in the first phases we only allow variables with very low level of incompatibility to be introduced and as we go, we allow more incompatible variables to be considered. A fast method is proposed to estimate the number of incompatibilities of a path while constructing it.

The advantages we gain through MPDCA can be summarized as follows:

1. The partition is disaggregated slowly.
2. It is more likely to introduce new degenerate variables resulting in further aggregation.
3. When an initial partition from a good feasible point (i.e. with a close-to-optimal objective value) is created, MPDCA usually finds close-to-optimal point in the first phase.
4. It is proven that the number of expected bases at each iteration is highly reduced because of the aggregation. This means less degenerate pivots and higher efficiency. However, since the partition changing process is time consuming, sometimes it may be more efficient to avoid aggregating the partition too often at the expense of executing some degenerate pivots [Elhallaoui et al., 2010b].

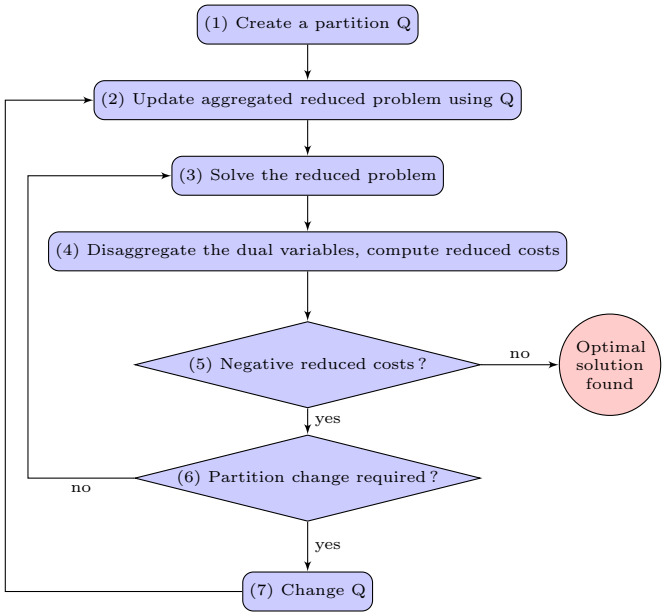



Figure 2.5 Dynamic Constraint Aggregation

MPDCA is reported to have a 4.5 average speedup relative to DCA and 23.4 relative to the standard column generation method Elhallaoui et al. [2010b].

### 2.2.2 Generalization to Linear Programming

The idea of constraint aggregation is generalized to the linear programming problems. Elhallaoui et al. [2010a] combine the work of Pan [1998] and Elhallaoui et al. [2008] to achieve a more efficient way to solve linear programming problems in general. The method is called Improved Primal Simplex (IPS). IPS starts with a feasible point, possibly acquired by a phase 1 Simplex algorithm. Afterwards, in the presence of degeneracy, it removes the constraints associated with degenerate variables. Suppose that we have $p$ non-degenerate variables. The $i$-th variable is said to be compatible with the current basis if the $i$-th column of $A$ is linearly dependent with the $p$ columns corresponding to the basis.

The process of disaggregation of the dual variables is done by solving a problem which is named Complementary Problem (CP). This problem is a generalization of the shortest path method used by Elhallaoui et al. in Elhallaoui et al. [2005], Elhallaoui et al. [2010b] and Elhallaoui et al. [2008]. It not only proposes a way to disaggregate the dual variables, but also provides us with some good variables with negative reduced costs. They also propose a method to change the basis at each iteration in such a way that each time the reduced problem is solved, a decrease in the objective function is guaranteed, or optimality or unboundedness

is concluded. If we have $m$ constraints and $p$ positive-valued variables in the basis, it is proven that entering at most $m - p + 1$ variables into the basis is sufficient to decrease the objective value or conclude unboundedness. These variables have two characteristics: first a convex combination of their corresponding columns in $A$ is linearly dependent with the columns of $A$ corresponding to the $p$ positive valued variables and second, the maximum reduced cost of the variable convex combination with regard to all the bases containing the $p$ positive-value variables is negative.

An acceleration strategy is also used in the solution process which is similar to the strategy used in the column generation process. At each iteration, more than one negative reduced cost variable is introduced to the master problem.

Raymond et al. [2010b] improve IPS to a more efficient method, called IPS2. The first improvement is to enforce a control on the number of variables entering the reduced problem (RP). IPS2 reduces the RP even further if it is possible and some conditions hold, e.g. when we perform multiple pivots without having significant effect on the objective function. The main idea of IPS2 is to remove dependent constraints from the RP. They propose a method that allows only independent rows to enter RP. The results show that on certain problems, IPS2 can perform 12 times faster than CPLEX.

### 2.2.3 Positive Edge

Inspired by IPS, Raymond et al. [2010a] propose another technique to take advantage of degeneracy which involves modifying the Simplex pivot rule, called the *positive edge* method. They propose an inexpensive procedure that determines compatible variables—a variable that upon entering the basis, causes an increase in the objective value. Compatibility of each variable is specified by performing a scalar product. We provide our account of the method in §3.2 and §4.3.

# CHAPTER 3

# ARTICLE 1: THE POSITIVE EDGE PIVOT RULE WITHIN COIN-OR's CLP

Mehdi Towhidi
École Polytechnique de Montréal & GERAD, Canada

Jacques Desrosiers
HEC Montréal & GERAD, Canada

François Soumis
École Polytechnique de Montréal & GERAD, Canada

## Abstract

This paper presents an implementation of the positive edge pivot rule using COIN-OR's CLP, where it has been combined with the Devex pricing criterion. Designed to take advantage of degeneracy in linear programming, a direct implementation leads to a better understanding of the full potential of the method. As a result, our implementation improves the performance of the positive edge method compared to the external implementation reported in the original paper, which uses an external implementation with CPLEX. To test its robustness, we also solved a set of linear problems from Mittelmann's library, which contains instances with a wide range of degeneracy levels. These computational results show that below a degeneracy level of 25%, the positive edge pivot rule is on average neutral while above this threshold, it reaches an average run time speedup of 2.3, with a maximum at 4.2 on an instance with a 75% degeneracy level.

## 3.1 Introduction

Introduced by Raymond et al. [2010], the *positive edge* pivot rule is designed to take advantage of degeneracy in linear programming. It acts as a filter and helps identifying so-called *compatible* variables that yield non-degenerate pivots in the primal Simplex algorithm. The identification follows from a simple calculation on the original column-vectors of the matrix of constraints and selecting such a variable with a negative reduced cost strictly improves the objective function value. Its computational complexity is the same as for the evaluation of the reduced cost.

The original implementation of the positive edge rule was done using the commercial solver CPLEX [1]. Commercial linear and mixed integer linear programming solvers allow customizing many pre-specified aspects of the solver's behavior using *callbacks*—references to user-defined functions. For example, users may define customized cuts or a specific branch-and-cut tree traversal strategy in integer programming, or determine termination conditions and printing out customized logs in linear programming, by coding their desired action and passing its reference for the solver to use when appropriate. However, typically, it is not possible to write and experiment with customized Simplex pivot rules. Instead, commercial softwares come with a set of predefined, and well optimized, pivot methods one can choose from. Among them are the smallest reduced cost, or Dantzig's rule [Dantzig, 1963], the steepest edge rule [Wolfe and Cutler, 1963, Goldfarb and Reid, 1977, Forrest and Goldfarb, 1992], and the Devex rule [Harris, 1975].

As a result, the positive edge rule was implemented in CPLEX using a workaround, which involves solving a partial problem with the help of two external procedures responsible of finding appropriate variables to be added to it (see §3.4.1 for details). Despite the incurred overhead, it demonstrates excellent performance on some benchmark problems, e.g., the PDS set [Carolan et al., 1990]. Nevertheless, it seems to struggle dealing with some others, e.g., the FOME instances [2]. A direct implementation of this new pivot rule leads to a better understanding of the full potential of the method. In this paper, we examine the efficiency of the positive edge rule by implementing it into COIN-OR's CLP [3]. Because, first, it obviously does not possess the limitations of commercial solvers explained above. Second, it has object-oriented structure, in C++ , which makes it relatively easy to understand and modify.

1. `http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/`
2. `http://www.gamsworld.org/performance/plib/`
3. `https://projects.coin-or.org/Clp`

This paper is organized as follows. First we present the positive edge rule in §3.2. In §3.3, we provide some implementation details. We present computational experiments in §3.4. Finally, we conclude in §3.5.

### 3.2 The positive edge pivot rule

Consider a linear program (LP) in standard form:

$$\underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} \quad \mathbf{c}^\top\mathbf{x} \quad \text{s.t.} \qquad \mathbf{A}\mathbf{x}=\mathbf{b}, \qquad \mathbf{x}\geq\mathbf{0}, \tag{3.1}$$

where $\mathbf{c}\in\mathbb{R}^n$, $\mathbf{A}\in\mathbb{R}^m\times\mathbb{R}^n$, $\mathbf{b}\in\mathbb{R}^m$, and $m<n$. Let $\mathrm{N}:=\{1,\ldots,n\}$ and $\mathrm{M}:=\{1,\ldots,m\}$. We denote by $\mathbf{0}$ a vector or a matrix with null entries of appropriate contextual dimensions. If B is a subset of an (ordered) index set, $\mathbf{x}_{\mathrm{B}}$ denotes the sub-vector of $\mathbf{x}$ indexed by B. Similarly, we denote by $\mathbf{A}_{\mathrm{RC}}$ the $|\mathrm{R}|\times|\mathrm{C}|$ submatrix of $\mathbf{A}\equiv\mathbf{A}_{\mathrm{MN}}$ whose rows and columns are indexed by R and C, respectively, while $\mathbb{R}_0:=\mathbb{R}\setminus\{0\}$ is the set of non-zero real numbers.

Let $\mathbf{A}_{\mathrm{B}}$ be the basis matrix for LP, where B denotes the index set of the basic variables. Define $\boldsymbol{Q}:=\mathbf{A}_{\mathrm{B}}^{-1}$. Let $\mathbf{x}_{\mathrm{B}}=\boldsymbol{Q}\mathbf{b}=\bar{\mathbf{b}}$ be a degenerate solution with $1\leq p<m$ non-zero variables. Define $\mathrm{P}:=\{i\mid\bar{b}_i>0\}$ as the index set of the $p$ rows where the positive (or non-degenerate) variables appear as basic and $\mathrm{Z}:=\{i\mid\bar{b}_i=0\}$ as the index set of the $m-p$ rows corresponding to the degenerate basic variables. Partition the inverse basis $\boldsymbol{Q}$ as $\begin{bmatrix}\boldsymbol{Q}_{\mathrm{PM}}\\ \boldsymbol{Q}_{\mathrm{ZM}}\end{bmatrix}$ such that $\bar{\mathbf{b}}_{\mathrm{P}}=\boldsymbol{Q}_{\mathrm{PM}}\mathbf{b}>\mathbf{0}$ gives the value of the positive variables while $\bar{\mathbf{b}}_{\mathrm{Z}}=\boldsymbol{Q}_{\mathrm{ZM}}\mathbf{b}=\mathbf{0}$ is for the degenerate basic variables. Let $\mathbf{a}_j=[a_{ij}]_{i\in\mathrm{M}}$ be the $j$-th column of $\mathbf{A}$ and $\bar{\mathbf{a}}_j=\boldsymbol{Q}\mathbf{a}_j$, the updated column-vector in the Simplex tableau: $\bar{\mathbf{a}}_j=[\bar{a}_{ij}]_{i\in\mathrm{M}}=\begin{bmatrix}\bar{\mathbf{a}}_{\mathrm{P}j}\\ \bar{\mathbf{a}}_{\mathrm{Z}j}\end{bmatrix}=\begin{bmatrix}\boldsymbol{Q}_{\mathrm{PM}}\\ \boldsymbol{Q}_{\mathrm{ZM}}\end{bmatrix}\mathbf{a}_j$.

**Definition 1.** *A variable $x_j, j\in\mathrm{N}$, is* compatible with the row-set P *if and only if* $\bar{\mathbf{a}}_{\mathrm{Z}j}=\mathbf{0}$.

A variable $x_j$ is compatible with the row-set P if and only if all $m-p$ components of $\bar{\mathbf{a}}_j$ are zero in the row-set Z [Raymond et al., 2010]. A variable $x_j$ for which $\bar{\mathbf{a}}_{\mathrm{Z}j}\neq\mathbf{0}$ is said to be *incompatible*. Positive basic variables are compatible whereas degenerate ones are not. Non-basic compatible variables yield non-degenerate pivots in the primal Simplex algorithm. Indeed, if $x_j$ is such a variable, the step size $\rho_j$, given by the *ratio-test*, is strictly positive and only needs to be computed over the row-set P: $\rho_j=\min\limits_{i\in\mathrm{P}|\bar{a}_{ij}>0}\left\{\dfrac{\bar{b}_i}{\bar{a}_{ij}}\right\}>0$. Hence when variable $x_j$ is selected to enter into the basis, the objective function strictly decreases by $\rho_j\bar{c}_j$ if $\bar{c}_j<0$, unless $\bar{a}_{ij}\leq 0$, $\forall i\in\mathrm{P}$, in which case it is unbounded.

The identification of compatible variables using Definition 1 requires the computation of the transformed matrix $\bar{\mathbf{A}}_{\text{ZN}} = \boldsymbol{Q}_{\text{ZB}}\mathbf{A}$. For large-scale problems, this can be time consuming. However, the *positive edge* rule allows to know if a variable $x_j$ is compatible or not *without explicitly computing* $\bar{\mathbf{a}}_{\text{Z}j}$. The criterion is based on the following observations. Let $\mathbf{v} \in \mathbb{R}_0^{m-p}$ be a random vector for which all components are different from zero. If $x_j$ is compatible, then $\bar{\mathbf{a}}_{\text{Z}j} = \mathbf{0}$ and hence $\mathbf{v}^\top \bar{\mathbf{a}}_{\text{Z}j} = \mathbf{0}$. Otherwise $x_j$ is incompatible ($\bar{\mathbf{a}}_j^{\text{z}} \neq 0$) and

$$\mathbf{v}^\top \bar{\mathbf{a}}_{\text{Z}j} = 0 \;\text{ if and only if }\; \mathbf{v} \perp \bar{\mathbf{a}}_{\text{Z}j}, \tag{3.2}$$

that is, if and only if the random vector $\mathbf{v} \in \mathbb{R}_0^{m-p}$ and $\bar{\mathbf{a}}_{\text{Z}j}$ are orthogonal, which is unlikely to happen. Let $\mathbf{w}^\top := \mathbf{v}^\top \boldsymbol{Q}_{\text{ZB}}$. Then $\forall j \in \text{N}$, $\mathbf{v}^\top \bar{\mathbf{a}}_{\text{Z}j} = \mathbf{v}^\top \boldsymbol{Q}_{\text{ZB}} \mathbf{a}_j = \mathbf{w}^\top \mathbf{a}_j$, and one can use $\mathbf{w}^\top \mathbf{a}_j$ for a compatibility-test using the original column-vector $\mathbf{a}_j$. This is similar to the replacement of $\mathbf{c}_{\text{B}}^\top \bar{\mathbf{a}}_j$ by $\boldsymbol{\pi}^\top \mathbf{a}_j$ in the computation of the reduced cost of variable $x_j$, where $\boldsymbol{\pi}^\top := \mathbf{c}_{\text{B}}^\top \mathbf{A}_{\text{B}}^{-1} = \mathbf{c}_{\text{B}}^\top \boldsymbol{Q}$ is the vector of dual variables associated with constraint set $\mathbf{Ax} = \mathbf{b}$.

**Positive edge criterion.** Let $\mathbf{v} \in \mathbb{R}_0^{m-p}$ be a random vector for which all components are different from zero. Define $\mathbf{w}^\top := \mathbf{v}^\top \boldsymbol{Q}_{\text{ZB}}$. The variable $x_j, j \in \text{N}$, is declared *compatible with the row-set* P if and only if $\mathbf{w}^\top \mathbf{a}_j = \mathbf{0}$.

There are several ways to integrate the positive edge criterion in a primal Simplex algorithm. The following one uses a two-dimensional rule: for a variable $x_j$, $j \notin \text{B}$, the first dimension computes the usual reduced cost $\bar{c}_j = c_j - \boldsymbol{\pi}^\top \mathbf{a}_j$, whereas the second evaluates $\mathbf{w}^\top \mathbf{a}_j$. The positive edge criterion identifies the index set of compatible variables $\text{C}_\mathbf{w} = \{j \notin \text{B} | \mathbf{w}^\top \mathbf{a}_j = 0\}$ and the index set of incompatible ones $\text{I}_\mathbf{w} = \{j \notin \text{B} | \mathbf{w}^\top \mathbf{a}_j \neq 0\}$. Notice that considering $\text{C}_\mathbf{w}$ is solely from a theoretical point of view, while in practice the compatibility of a variable is only checked for the variables with a negative reduced cost or a subset of them. Let $\bar{c}_{j_\star}$ be the smallest reduced cost indexed by $j_\star \in \text{C}_\mathbf{w} \cup \text{I}_\mathbf{w}$ and let $\bar{c}_{j_\mathbf{w}}$ be the smallest reduced cost for a compatible variable, indexed by $j_\mathbf{w} \in \text{C}_\mathbf{w}$. The current solution is optimal if $\bar{c}_{j_\star} \geq 0$. We initialize both $\bar{c}_{j_\star}$ and $\bar{c}_{j_\mathbf{w}}$ at zero. Compatible variables with negative reduced cost should be preferred to enter the basis except if $\bar{c}_{j_\star}$ is much smaller than $\bar{c}_{j_\mathbf{w}}$. This can be controlled with a threshold parameter $0 \leq \psi < 1$. Hence the selection rule becomes:

*Two-dimensional positive edge selection rule*:
**if** $\bar{c}_{j_\star} < 0$ and $\bar{c}_{j_\mathbf{w}} < \psi \bar{c}_{j_\star}$, **then** select $x_{j_\mathbf{w}}$ **otherwise**, select $x_{j_\star}$.

Naturally, the above rule can be used in the context of a partial pricing strategy. It can also be combined with the Devex pricing criterion. Finally, small numerical errors may occur on a

computer. However, the consequence of selecting an incompatible variable which is identified as compatible is not important: this simply results in a degenerate pivot, the same situation as if an incompatible variable $x_{j_\star}$, $j_\star \in \mathrm{I}_{\mathbf{w}}$ were selected to enter the basis.

## 3.3 Implementation

In this section, we first present in §3.3.1 some concerns regarding the implementation of the positive edge rule. In §3.3.2, we provide low-level details of the code behind the implementation.

### 3.3.1 Implementation details

Positive edge is designed to reduce the computing time of the primal Simplex algorithm by reducing the number of iterations. This comes at the cost of adding a time-overhead to an average iteration, while aiming to perform considerably fewer iterations. A significant contributor to this overhead is the time necessary to *update the partition*—redefining the row-sets P and Z based on the zero (or degenerate) values in the current basic solution point, generating a random vector $\mathbf{v}$, and computing $\mathbf{w}^\top = \mathbf{v}^\top \boldsymbol{Q}_{\mathrm{ZB}}$. Therefore, the question is, how often should we update the partition ?

At a given iteration, when we choose a variable that is compatible with the row-set P as the entering variable, say $x_j$, then $\bar{\mathbf{a}}_{\mathrm{Z}j} = \mathbf{0}$, hence $\bar{\mathbf{b}}_{\mathrm{Z}}$ remains at zero after the pivot and the same partition can be reused in the next iteration. Note that in this situation we might have added zero entries to $\bar{\mathbf{b}}_{\mathrm{P}}$ which subsequently changes $\bar{\mathbf{b}}_{\mathrm{Z}}$. This is one case in which a partition update is required. On the other hand, if we choose an incompatible variable, we perform a degenerate pivot and we stay at the same solution point. However, in actual problems, after carrying out multiple degenerate pivots, we eventually find a basis that allows for a step that improves the objective value. This step can potentially change the index sets P and Z substantially, making it another case that calls for a partition update. To detect these cases, we consider a sudden change in the number of positive elements in the updated right-hand-side $\bar{\mathbf{b}}$ as a sign of partition update requirement. We call this sudden change a *jump*.

To this end, after $l$ pivots, where $l$ is a parameter, we check the number of positive elements $p$ in $\bar{\mathbf{b}}$ and form sets P and Z. If $|p - p_{old}| > k$, where $k$ is a positive integer and $p_{old}$ is the stored value of the number of positive basic variables from the most recent updated row partition, then we say we have a jump and we need to update the partition. Smaller values

of $k$ cause more frequent and possibly unnecessary updates while large values of $k$ may result in incorrect recognition of compatible or incompatible variables, since the partition does not reflect the current status of $\bar{\mathbf{b}}$. Therefore, there is a trade-off.

In practice, while solving an LP, we observe that there are periods during which the number of positive variables remains nearly constant, while in other stages this number is more volatile. Therefore, parameter $l$ should be adjusted dynamically depending on the rate of change of $p$. For example, we start by forming $\bar{\mathbf{b}}_{\mathrm{P}}$ at every $l = 100$ iterations and we look for jumps with $k = 10$. If there are no jumps, we set $l := min\{300, l + 50\}$, where 300 is the maximum acceptable value for $l$. On the other hand, if a jump is detected, we update $l$ to be $l := max\{50, l - 50\}$. This strategy allows the method to adapt—update the partition less frequently when it seems that we are updating too often. On the contrary, we update more often when a jump is identified, signaling that we are likely to update at a slow pace. Note that using the proposed updating formulas, we always maintain $l$ in a reasonable range, namely, in our tests $50 \leq l \leq 300$.

Regarding the choice of the random vector $\mathbf{v}$, we have the following, according to Raymond et al. [2010]. The random vector $\mathbf{v} \in \mathbb{R}_0^{m-p}$ is chosen such that all components are independent and identically distributed: $v_i \sim SEM_{32}, i \in \mathrm{Z}$. Definition and some properties of the $SEM_{32}$ distribution follows.

**Definition 2.** *A floating-point number $F$ with distribution $SEM_{32}$ is a single precision number where the sign-bit $S$, the exponent field $E$, and the mantissa field $M$ are independent and follow the discrete uniform distributions $S \sim U[0, 1]$, $E \sim U[64, 191]$, and $M \sim U[0, 2^{23} - 1]$.*

With $SEM_{32}$ symmetric around zero, $\mu_F = 0$ while we derive next a lower bound on the standard deviation $\sigma_F$. Let $V$ be a random variable following a $SEM_{32}$ distribution but without considering the mantissa field. Then, for the exponent field, we have $-63 \leq E-127 \leq 64$, and hence $V \in \{\pm 2^{-63}, \pm 2^{-62}, \ldots, \pm 2^{64}\}$ with equiprobable events. Therefore

$$\sigma_F^2 > \sigma_V^2 = \frac{2}{2^8}(\sum_{k=1}^{63}(2^{-k})^2 + \sum_{k=0}^{64}(2^k)^2) > \frac{2}{2^8}\sum_{k=0}^{64}(2^2)^k.$$

Let $S_{n+1}(x) = \sum_{k=0}^{n+1} x^k$. Then $S_{n+1}(x) = 1 + xS_n(x)$ and $S_{n+1}(x) = S_n(x) + x^{n+1}$. Hence, for $1 \neq x \in \mathbb{R}$, $S_n(x) = \frac{x^{(n+1)}-1}{x-1}$ and

$$\sigma_F^2 > \frac{2}{2^8}S_{64}(2^2) = \frac{2}{2^8} \times \frac{2^{2(64+1)} - 1}{2^2 - 1} = \frac{1}{2^8} \times \frac{2}{3} \times (2^{2(64)} \times 4 - 1) > \frac{2^{2(64)}}{2^8} = 2^{120}.$$

Hence, $\sigma_F > 2^{60}$. The discrete distribution $SEM_{32}$ is symmetric around zero with a huge dispersion. Finally, in our implementation, we use a vector $\mathbf{v}$ of dimension $m$ where we set $v_i = 0, \forall i \in \mathrm{P}$ whereas $v_i \sim SEM_{32}, \forall i \in \mathrm{Z}$.

### 3.3.2 Implementing the positive edge pivot rule within CLP

CLP is an open-source LP solver and a part of the COIN-OR [4] collection. It is equipped with Simplex and barrier methods. The solver is coded in C++ with an object-oriented design. This makes it a perfect choice to implement the positive edge rule.

Figure 3.1 shows a partial Unified Modeling Language (UML) Class diagram [Larman, 2001] of CLP. It considers a section of CLP's structure that is involved in the process of defining pivots. The `ClpModel` class is where the essential information and operations regarding a linear optimization model are defined. Its `ClpSimplex` subclass contains more Simplex-related structures. Derived from `ClpSimplex` are `ClpSimplexPrimal` and `ClpSimplexDual` implementing necessary methods for their respective algorithms.

`ClpSimplex` has an attribute of type `ClpPrimalPivotColumn`, displayed by an aggregation relationship. The latter is the base class for pivots in CLP. For example, the `ClpPrimalColumnDantzig` and `ClpPrimalColumnSteepest` classes, which are predefined in CLP, both derive from `ClpPrimalPivotColumn`.

Pivots must implement a virtual method, `pivotColumn()`, which returns an integer specifying the index of the variable selected as the entering variable. Similarly, to define the positive edge pivot rule in CLP, we must subclass `ClpPrimalPivotColumn` and implement the rule in the `pivotColumn()` method.

## 3.4 Computational Experiments

In §3.4.1, we compare our internal CLP results with those of Raymond et al. [2010], where the authors use an external CPLEX implementation on different test sets, including PDS [Carolan et al., 1990] and FOME [5] instances. In §3.4.2, to see how CLP equipped with positive edge performs on a general set of LPs, we run tests on relatively large instances of Mittelmann's LP benchmark [6], i.e., those that are reported to take more than 10 seconds

4. `http://www.coin-or.org/`
5. `http://plato.asu.edu/ftp/lptestset/fome/`, (Feb 2011)
6. `http://plato.asu.edu/ftp/lpcom.html`

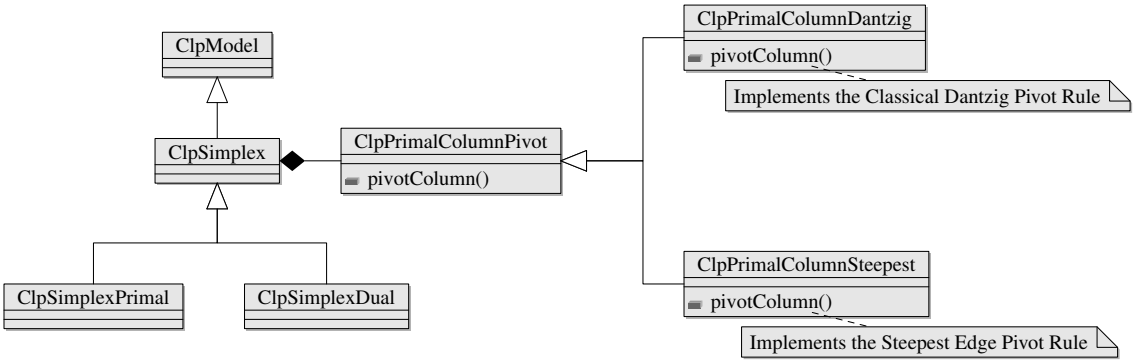



Figure 3.1 Partial UML Class Diagram for CLP

to solve using CPLEX. We perform all our experiments using computers with Intel Xeon 2.4GHz quad-core processors.

During the tests, we consider incorporating positive edge into Dantzig's pivot rule, as explained in §3.2, and the Devex rule, which is also very similar. In the former, we choose $\psi = 0.1$, while in the latter we set $\psi = 0.5$. The reason is that the Devex rule already avoids degenerate pivots, or tiny steps, to some extent. Neither of these methods show considerable sensitivity to small changes in the value of $\psi$.

### 3.4.1 A comparison between the external and CLP implementations

In this section, to test the efficiency of the CLP implementation of positive edge we choose two of the same sets of problems used by Raymond et al. [2010] to analyze their external CPLEX implemention, the PDS and FOME instances. They are relatively sparse and degenerate, as shown in Table 3.1. The degeneracy level is measured as the average over all iterations of $1 - p/m$, that is, the number of zero basic variables divided by the number of constraints. Average degeneracy is only 40% for the FOME problems while it reaches 75% for the PDS ones.

First of all, we verified that the classical Dantzig pivot rule performance does not come close to the Devex pricing rule within CLP. The results for two instances, PDS-50 and PDS-60, are reported in Table 3.2: the Devex criteria is more than 15 times faster. However, the reader can observe that the use of the positive edge strongly improves the Dantzig's rule performance on these instances: the number of iterations (pivots) is approximately reduced by a factor of 10 while the cpu time is divided by 8. All results reported in the subsequent tests are obtained using the Devex pricing rule of CLP.

Table 3.1 PDS and FOME instances

| Problem | $n$:Vars | $m$:Cons | non-zeros | Sparsity | Degeneracy |
|---|---|---|---|---|---|
| PDS-20 | 105728 | 33874 | 230200 | 6.43E-05 | 0.69 |
| PDS-30 | 154998 | 49944 | 337144 | 4.36E-05 | 0.72 |
| PDS-40 | 212859 | 66844 | 462128 | 3.25E-05 | 0.73 |
| PDS-50 | 270095 | 83060 | 585114 | 2.61E-05 | 0.74 |
| PDS-60 | 329643 | 99431 | 712779 | 2.17E-05 | 0.74 |
| PDS-70 | 382311 | 114944 | 825771 | 1.88E-05 | 0.74 |
| PDS-80 | 426278 | 129181 | 919524 | 1.67E-05 | 0.75 |
| PDS-90 | 466671 | 142823 | 1005359 | 1.51E-05 | 0.75 |
| PDS-100 | 505360 | 156243 | 1086785 | 1.38E-05 | 0.77 |
| FOME12 | 48920 | 24284 | 142528 | 1.20E-04 | 0.40 |
| FOME13 | 97840 | 48568 | 285056 | 6.00E-05 | 0.38 |

Table 3.2 Dantzig's rule vs. Devex pricing

| | CLP | | | | CLP + Positive Edge | | | |
|---|---|---|---|---|---|---|---|---|
| | *Dantzig's rule* | | *Devex pricing* | | *Dantzig's rule* | | *Devex pricing* | |
| Problem | pivots | time (s) | pivots | time (s) | pivots | time (s) | pivots | time (s) |
| PDS-50 | 2631164 | 9536.07 | 238961 | 579.98 | 224622 | 1208.61 | 74708 | 155.77 |
| PDS-60 | 3922444 | 20045.60 | 311517 | 940.59 | 397185 | 2730.28 | 107666 | 273.64 |

For the next comparisons, we use the notion of *speedup* regarding the number of iterations or the cpu time. For example, the ratio $t_{clp}/t_{pe}$ in Table 3.3 computes the time speedup, where $t_{clp}$ is the run time of the CLP's solver, and $t_{pe}$ is the execution time of the primal Simplex solver equipped with the positive edge rule. The results are presented in Table 3.3, where the performance of an external CPLEX implementation [Raymond et al., 2010] appears on the right-hand side. Essentially, this external implementation works as follows: Given a degenerate solution, the primal Simplex algorithm of CPLEX solves a partial problem that is supplied by two external procedures: the first identifies all compatible variables to be included in the partial problem, if any, otherwise the second brings in all variables with negative reduced costs.

Table 3.3 Positive Edge implementations: CLP vs. external (CPLEX)

| | CLP | | CLP + Positive Edge | | | | External (CPLEX) |
|---|---|---|---|---|---|---|---|
| Problem | pivots $i_{clp}$ | time $t_{clp}$ | pivots $i_{pe}$ | time $t_{pe}$ | speedup $i_{clp}/i_{pe}$ | speedup $t_{clp}/t_{pe}$ | time speedup of Raymond et al. [2010] |
| PDS-20 | 14660 | 7.03 | 11623 | 5.97 | 1.26 | 1.18 | 0.83 |
| PDS-30 | 33622 | 31.63 | 25158 | 20.87 | 1.34 | 1.52 | 0.84 |
| PDS-40 | 91710 | 157.91 | 49210 | 71.64 | 1.86 | 2.20 | 1.42 |
| PDS-50 | 238961 | 579.98 | 74708 | 155.77 | 3.20 | 3.72 | 1.92 |
| PDS-60 | 311517 | 940.59 | 107666 | 273.64 | 2.89 | 3.44 | 1.95 |
| PDS-70 | 469283 | 1700.26 | 152027 | 508.01 | 3.09 | 3.35 | 1.92 |
| PDS-80 | 601199 | 2625.82 | 173245 | 620.70 | 3.47 | 4.23 | 1.68 |
| PDS-90 | 597643 | 2645.29 | 211421 | 875.76 | 2.83 | 3.02 | 1.86 |
| PDS-100 | 597181 | 2717.60 | 196559 | 856.23 | 3.04 | 3.17 | 1.55 |
| | | **PDS average speedup** | | | **2.55** | **2.87** | **1.55** |
| FOME12 | 172518 | 228.80 | 165006 | 150.46 | 1.05 | 1.52 | 0.16 |
| FOME13 | 433810 | 1339.79 | 391839 | 526.73 | 1.11 | 2.54 | 0.09 |
| | | **FOME average speedup** | | | **1.08** | **2.05** | **0.13** |
| | | **average speedup** | | | **2.29** | **2.72** | **1.29** |

First, we observe how our implementation improves the performance of the positive edge method compared to the external implementation. The average time speedup of 1.29, reported by Raymond et al. [2010], rises significantly to 2.72. More specifically, we obtain a speedup of 1.52 and 2.54 on the FOME instances, compared to 0.16 and 0.09 of the external CPLEX implementation. Similarly, on smaller problems of the PDS set, PDS-20 and PDS-30, we now have a more-than-one speedup factor. Finally, the maximum speedup of our implementation on these test problems is 4.23, considerably larger than 1.95 in the external implementation.

Regarding the comparison with CLP, the average time speedup is 2.72, a bit larger than the reduction factor of 2.29 for the number of pivots. Similar results for the PDS instances are observed, respectively 2.87 for the time speedup and 2.55 for the number of iterations. Finally, although the number of pivots is about the same for the FOME instances with a speedup factor of only 1.08, the CLP implementation with the positive edge selection rule is more than two times faster (2.05).

### 3.4.2 Results on Mittelmann's LP instances

Table 3.4 presents dimensional aspects, degeneracy level, and sparsity level of the selected linear problems of Mittelmann's LP test set. These problems are sorted by their level of degeneracy. In Table 3.1, we have already provided the statistics on problems PDS-40, PDS-100, FOME12, and FOME13, which also belong to this test set.

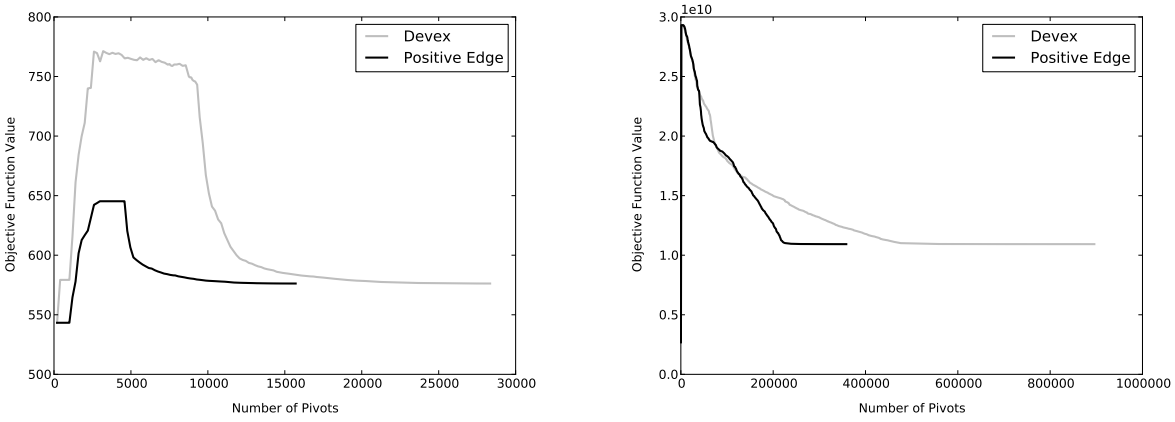
Table 3.4 Mittelmann's LP instances

| Problem | $n$:Vars | $m$:Cons | non-zero | Sparsity | Degeneracy |
|---|---|---|---|---|---|
| cont4 | 40398 | 160793 | 398399 | 6.13E-05 | 0 |
| cont1 | 40398 | 160793 | 399991 | 6.16E-05 | 6.57E-007 |
| self | 7364 | 960 | 1148845 | 1.63E-01 | 0.002 |
| stat96v4 | 62212 | 3173 | 490472 | 2.48E-03 | 0.03 |
| rail4284 | 1092610 | 4284 | 12372358 | 2.64E-03 | 0.04 |
| neos | 36786 | 479120 | 1084461 | 6.15E-05 | 0.12 |
| lp22 | 13434 | 2959 | 78994 | 1.99E-03 | 0.19 |
| neos1 | 1892 | 131582 | 468094 | 1.88E-03 | 0.19 |
| rlfprim | 8052 | 57422 | 264483 | 5.72E-04 | 0.20 |
| mod2 | 31728 | 35665 | 220116 | 1.95E-04 | 0.23 |
| stat96v1 | 197472 | 5995 | 588798 | 4.97E-04 | 0.23 |
| nug08-3rd | 20448 | 19728 | 139008 | 3.45E-04 | 0.24 |
| world | 32734 | 35511 | 220748 | 1.90E-04 | 0.24 |
| neos2 | 1560 | 132569 | 552596 | 2.67E-03 | 0.29 |
| dano3mip | 13873 | 3203 | 79656 | 1.79E-03 | 0.35 |
| nug15 | 22275 | 6331 | 110700 | 7.85E-04 | 0.36 |
| neos3 | 6624 | 512209 | 1542816 | 4.55E-04 | 0.50 |
| watson_2 | 671861 | 352014 | 1843716 | 7.80E-06 | 0.83 |
| ns1688926 | 16587 | 32768 | 1712128 | 3.15E-03 | 0.86 |
| dbic1 | 183235 | 43200 | 1038761 | 1.31E-04 | 0.88 |

Table 3.5 shows the numerical results. Again, the problems are sorted from lower to higher levels of degeneracy. One the one hand, for all problems with a degeneracy level of less than 25%, the positive edge pivot rule is useless on average, with speedup factors of 1.01 for the number of pivots and 1.00 for the run time. The worst outcome belongs to `rail4284`, with a 4% degeneracy level, where positive edge requires twice the run time to find an optimal solution. Of course, one can avoid such *slowdowns* by checking the degeneracy level of a problem and decide whether it is beneficial or not to use the positive edge rule.

On the other hand, for the 11 instances with a degeneracy level of at least 25%, the time speedup is always greater than one, with an average value reaching 1.97 while the number of pivots decreases by an average factor of 1.67. Moreover, 5 of these instances show a run time reduction by more than 200%. In the case of `ns1688926`, marked by a star in the table, for the positive edge test we had to change CLP's default primal feasibility tolerance from $10^{-7}$ to $10^{-5}$, otherwise the solution process struggles to obtain primal and dual feasibility at the same time.

Figure 3.2 illustrates the effect of the positive edge rule on reducing the number of required pivots to obtain an optimal solution. In the case of `dano3mip`, shown in Figure 3.2(a), we perform 46% less pivots, while in that of `PDS-100`, Figure 3.2(b), we cut the number of pivots by a factor of 3. Observe that in Figure 3.2(a), the objective function at the beginning of the solution process is less than the optimal objective value. This occurs because those iterations are primal infeasible.




(a) dano3mip (b) PDS-100

Figure 3.2 Comparing the number of pivots in CLP: Positive Edge vs. Devex

Finally, in Figure 3.3, we plot the time speedup against the level of degeneracy for PDS [Carolan et al., 1990] and Mittelmann's LP instances. It demonstrates that on those ins-

Table 3.5 Positive Edge on Mittelmann's instances

| Problem | CLP | | CLP + Positive Edge | | | |
|---|---|---|---|---|---|---|
| | pivots $i_{clp}$ | time $t_{clp}$ | pivots $i_{pe}$ | time $t_{pe}$ | speedup $i_{clp}/i_{pe}$ | speedup $t_{clp}/t_{pe}$ |
| cont4 | 40602 | 593.54 | 40602 | 552.55 | 1 | 1.07 |
| cont1 | 41283 | 685.97 | 41283 | 750.12 | 1 | 0.91 |
| self | 12119 | 196.31 | 12119 | 181.43 | 1 | 1.08 |
| stat96v4 | 186295 | 582.35 | 190660 | 635.49 | 0.98 | 0.92 |
| rail4284 | 801367 | 47762.70 | 1370343 | 105049.00 | 0.58 | 0.45 |
| neos | 194690 | 1068.16 | 183044 | 1000.06 | 1.06 | 1.07 |
| lp22 | 51655 | 64.36 | 39059 | 34.97 | 1.32 | 1.84 |
| neos1 | 3588 | 18.47 | 3286 | 17.57 | 1.09 | 1.05 |
| rlfprim | 3401 | 2.30 | 3088 | 2.49 | 1.10 | 0.92 |
| mod2 | 177939 | 337.00 | 181270 | 359.80 | 0.98 | 0.94 |
| stat96v1 | 43534 | 253.36 | 44054 | 303.76 | 0.99 | 0.83 |
| nug08-3rd | 375000 | 27287.00 | 429959 | 31110.70 | 0.87 | 0.88 |
| world | 213475 | 389.64 | 193750 | 395.48 | 1.10 | 0.99 |
| **Degeneracy level $<$ 25%: average speedup** | | | | | **1.01** | **1.00** |
| neos2 | 6464 | 46.81 | 6539 | 45.33 | 0.99 | 1.03 |
| dano3mip | 26746 | 26.99 | 18373 | 16.11 | 1.46 | 1.68 |
| nug15 | 137087 | 847.48 | 117870 | 721.83 | 1.16 | 1.17 |
| FOME13 | 631758 | 1339.79 | 391839 | 526.73 | 1.61 | 2.54 |
| FOME12 | 221658 | 228.80 | 165006 | 150.46 | 1.34 | 1.52 |
| neos3 | 769771 | 34302.50 | 554988 | 23435.30 | 1.39 | 1.46 |
| PDS-40 | 91710 | 157.91 | 49210 | 71.64 | 1.86 | 2.20 |
| PDS-100 | 597181 | 2717.60 | 196559 | 856.23 | 3.04 | 3.17 |
| watson_2 | 270607 | 747.13 | 232001 | 412.81 | 1.17 | 1.81 |
| ns1688926* | 48061 | 744.77 | 18497 | 256.24 | 2.60 | 2.91 |
| dbic1 | 30215 | 86.41 | 17733 | 40.15 | 1.70 | 2.15 |
| **Degeneracy level $\geq$ 25%: average speedup** | | | | | **1.67** | **1.97** |

tances, positive edge rule becomes more effective as the degeneracy level increases. Below a degeneracy level of 25%, the method is on average neutral as reported in Table 3.5. Above this threshold, it reaches an average run time speedup of 2.31 on the combined results of Table 3.3 and Table 3.5, with a maximum at 4.23 on PDS-80, an instance with a 75% degeneracy level.

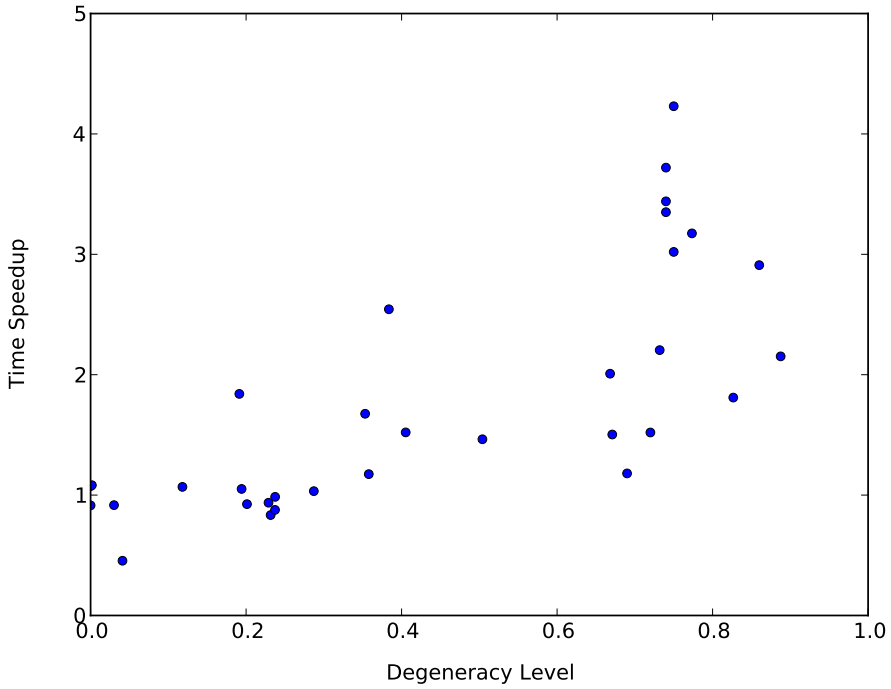



Figure 3.3 Effect of the degeneracy level on the time speedup

## 3.5 Conclusion

In this paper, we presented an efficient implementation of the positive edge pricing rule within CLP, an open source implementation of the Simplex method for linear programming. Designed to take advantage of degeneracy, it acts as a filter that identifies non-basic variables and yields non-degenerate pivots in the primal Simplex algorithm. Tested on PDS [Carolan et al., 1990] and Mittelmann's LP instances, the computational results show that below a degeneracy level of 25%, this pricing rule is on average neutral, while above this threshold, the average run time of more than two times faster. On specific highly degenerate instances, namely PDS and FOME test sets, CLP implementation results in a maximum speedup of more than 4, comparing to a maximum speedup of 2 for the external CPLEX implementation on the same problems.

## REFERENCES

W. Carolan, J. Hill, J. Kennington, S. Niemi, and S. Wichmann. Empirical Evaluation of the KORBX Algorithms for Military Airlift Applications. *Operations Research*, 38:240–248, 1990.

G. B. Dantzig. *Linear programming and extensions.* Princeton University Press, New Jersey, 1963.

J. J. Forrest and D. Goldfarb. Steepest-edge Simplex algorithms for linear programming. *Mathematical Programming*, 57:341–374, 1992. ISSN 0025-5610. DOI: 10.1007/BF01581089.

D. Goldfarb and J. K. Reid. A practicable steepest-edge Simplex algorithm. *Mathematical Programming*, 12:361–371, 1977. ISSN 0025-5610. DOI: 10.1007/BF01593804.

P. M. J. Harris. Pivot selection methods of the Devex LP code. In R. W. Cottle, L. C. W. Dixon, B. Korte, M. J. Todd, E. L. Allgower, W. H. Cunningham, J. E. Dennis, B. C. Eaves, R. Fletcher, D. Goldfarb, J.-B. Hiriart-Urruty, M. Iri, R. G. Jeroslow, D. S. Johnson, C. Lemarechal, L. Lovasz, L. McLinden, M. J. D. Powell, W. R. Pulleyblank, A. H. G. Rinnooy Kan, K. Ritter, R. W. H. Sargent, D. F. Shanno, L. E. Trotter, H. Tuy, R. J. B. Wets, E. M. L. Beale, G. B. Dantzig, L. V. Kantorovich, T. C. Koopmans, A. W. Tucker, P. Wolfe, M. L. Balinski, and Eli Hellerman, editors, *Computational Practice in Mathematical Programming*, volume 4 of *Mathematical Programming Studies*, pages 30–57. Springer Berlin Heidelberg, 1975. ISBN 978-3-642-00766-8. DOI: 10.1007/BFb0120710.

C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, 2nd Edition.* Prentice Hall, 2001.

V. Raymond, F. Soumis, A. Metrane, and J. Desrosiers. Positive edge: A pricing criterion for the identification of non-degenerate Simplex pivots. Cahier du GERAD G-2010-61, GERAD, Montréal, Québec, Canada, 2010.

P. Wolfe and L. Cutler. Experiments in linear programming. In Graves and Wolfe, editors, *Recent Advances in Mathematical Programming.* McGraw-Hill, New York, 1963.

# CHAPTER 4

# ARTICLE 2: CUSTOMIZING THE SOLUTION PROCESS OF COIN-OR's LINEAR SOLVERS WITH PYTHON

Mehdi Towhidi
École Polytechnique de Montréal & GERAD, Canada

Dominique Orban
École Polytechnique de Montréal & GERAD, Canada

## Abstract

Implementations of the Simplex method differ only in very specific aspects such as the pivot rule. Similarly, most relaxation methods for mixed-integer programming differ only in the type of cuts and the exploration of the search tree. Implementing instances of those frameworks would therefore be more efficient if linear and mixed-integer programming solvers let users customize such aspects easily. We provide a scripting mechanism to easily implement and experiment with pivot rules for the Simplex method by building upon COIN-OR's open-source linear programming package CLP. Our mechanism enables users to implement pivot rules in the Python scripting language without explicitly interacting with the underlying C++ layers of CLP. In the same manner, it allows users to customize the solution process while solving mixed-integer linear programs using the CBC and CGL COIN-OR packages. The Cython programming language ensures communication between Python and COIN-OR libraries and activates user-defined customizations as callbacks. For illustration, we provide an implementation of a well-known pivot rule as well as the positive edge rule—a new rule that is particularly efficient on degenerate problems, and demonstrate how to customize branch-and-cut node selection in the solution of a mixed-integer program.

## 4.1 Introduction

The Simplex algorithm created by Dantzig is considered by many to be among the top ten algorithms of the twentieth century in terms of its scientific and practical impact [Cipra, 2000, Dongarra and Sullivan, 2000]. Simplex is a graceful way of solving linear programs

(LP). Although it is efficient in many situations, it has always struggled in the face of degeneracy, whose effects range from causing cycling to significantly impacting the performance. In order to ensure convergence theoretically one needs to modify an essential component of the algorithm—the *pivot selection* (e.g. Bland [1977]). Identifying a pivot selection rule that is efficient across many degenerate LP instances is still an open subject after more than 60 years of research. The Improved Primal Simplex (IPS) of Elhallaoui et al. [2010] and the positive edge pivot rule of Raymond et al. [2010] are recent efforts in that direction.

Commercial implementations of Simplex typically do not allow users to plug in customized pivot rules. Raymond et al. [2010] work around this limitation by solving auxiliary dynamically-generated LPs at each step—a substantial overhead. A promising alternative is to modify an open-source Simplex implementation, such as Makhorin's GLPK or COIN-OR's CLP, typically written in a low-level programming language such as C or C++ . Delving into large-scale open projects in such languages can be a daunting task even for a seasoned programmer. It does appear however that open-source implementations of Simplex are the ideal platform to implement and experiment with pivot rules.

In this paper we propose a user-friendly alternative that emphasizes flexibility and ease of use, and promotes fast development and productivity. CyLP is a tool for researchers to implement pivot rules in the dynamic high-level Python programming language [1]. CyLP builds upon CLP and provides flexible and easy to use mechanisms to substitute CLP's built-in pivot rules with a user-defined pivot rule written in Python. Aside from LP, effective pivot rules are also crucial in mixed-integer linear programming (MIP). CyLP provides facilities to customize the solution of MIPs using Python, by allowing users to inject cuts of their own design. In particular, we interface COIN-OR's CBC [2] which provides tools to solve MIPs using branch-and-cut [Hoffman and Padberg, 1993, Padberg and Rinaldi, 1991]. In addition, CyLP can be used as a modeling environment to formulate and solve LPs and MIPs via CLP and CBC. Our main motivation for this research is the design of pivot rules suited to degenerate problems. In follow-up research, we apply such rules to quadratic programs (QP) and mixed-integer QPs.

The pivot rule is part of the nerve center of any implementation of Simplex since it must be executed at each iteration. One worry is thus that implementing it in an interpreted language seriously affects performance. In order to limit the performance hit as much as possible, our choice is to write the communication layer between Python and CLP in the Cython [3] programming language. Cython is a strongly-typed superset of Python whose main design

1. `www.python.org`
2. `projects.coin-or.org/Cbc`
3. `www.cython.org`

goal is strictly speed and that eases the interfacing of binary code as well as of source code. This feature makes it particularly suitable for facilitating communication between Python and large libraries that users may not wish to recompile. CyLP is composed of three layers: a few C++ interface classes, a thin Cython layer ensuring fast communication between the C++ code and the Python programming language, and convenience Python classes. As our numerical experiments illustrate, not only are we able to maintain competitive execution speeds, but the gains in flexibility and ease of development far outweigh the performance hit. CyLP is available as an open-source package from `github.com/mpy/CyLP`.

The rest of this paper is organized as follows. Section 4.2 gives a brief description of the Simplex method, common pivot rules typically found in solvers and the need to define and examine new pivot rules. In Section 4.3 we present the positive edge method, specifically designed for degenerate problems. Section 4.4 describes some implementation details of CyLP, provides an implementation of Dantzig's classic pivot rule as an example and shows the essentials of our implementation of the positive edge rule in Python. It also covers how CyLP is used to solve MIPs, how it can be used to examine different solution strategies scripted in Python, and summarizes its modeling capabilities. Section 4.5 documents numerical experience. We conclude and look ahead in §4.6.

### Related Research

Two of the major commercial LP solvers, CPLEX [4] and Gurobi [5], offer a Python API and allow users to interact with the solution process of MIPs using callbacks to customize cut-generation and the branch-and-cut procedure. They do not appear to let users define pivot rules.

PuLP is a Python modeler for LP and provides interfaces to existing open-source and commercial LP solvers such as GLPK, CLP and CPLEX. PyCPX [Koepke, a] is a Cython interface to CPLEX that leverages the power of Numpy [6]—a library defining the standard array type in Python—and provides more convenient modeling facilities than the default CPLEX Python API. Pylpsolve [Koepke, b] is a similar interface to lpsolve [Berkelaar] and Pycoin [Silva, 2005] is a Python interface to CLP. None of them appears to allow users to customize the solution process.

4. `www.cplex.com`
5. `www.gurobi.com`
6. `www.numpy.org`

There is growing interest in Cython as an interface language for projects written in low-level languages. For example, CyIPOPT [Aides] is a wrapper for the interior-point optimizer IPOPT [Wächter and Biegler, 2006].

To the best of our knowledge, CyLP is the first toolset that connects with an efficient implementation of Simplex and permits experimentation with pivot rules in a high-level language. Like PyCPX, CyLP allows users to exploit the power of Numpy.

**Notation**

Throughout this paper we use capital latin letters for matrices and lowercase latin letters for vectors. Calligraphic letters are used to denote index sets. For any matrix $M$, we denote the $j$-th column of $M$ by $M_j$, the $i$-th row of $M$ by $M^i$ and the $i$-th element of $M_j$ by $m_{ij}$. For any vector $c$, any matrix $A$ and any index set $\mathcal{B}$, $c_{\mathcal{B}}$ is the subvector of $c$ indexed by $\mathcal{B}$ and $A_{\mathcal{B}}$ is the submatrix of $A$ composed of the columns indexed by $\mathcal{B}$. Similarly $A^{\mathcal{B}}$ is the submatrix of $A$ which contains the rows indexed by $\mathcal{B}$. The only norm used in this paper, denoted $\|\cdot\|$, is the Euclidian norm.

## 4.2 Implementing Simplex Pivot Rules

In this section, we give a high-level, and by no means complete, account of the Simplex method and an overview of the CLP implementation.

### 4.2.1 The Simplex Method

The linear programming problem in standard form is

$$\underset{x\in\mathbb{R}^n}{\text{minimize}}\ c^T x \quad \text{s.t.} \qquad Ax = b,\ x \geq 0, \tag{LP}$$

where $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$, $A \in \mathbb{R}^{m\times n}$ and the inequality $x \geq 0$ is understood elementwise. Simplex is an iterative method. It divides variables into two categories: basic and non-basic. Let $\mathcal{B}$ be the index set of basic variables, also called the *basis*, and $\mathcal{N}$ be that of the non-basic variables. At every Simplex iteration, $|\mathcal{B}| = m$ and $|\mathcal{N}| = n - m$. Non-basic variables are fixed to zero because if (LP) has a solution, then there exists a solution with at most $m$ nonzero

elements [Dantzig, 1963]. Using those index sets, we partition $A$, $x$, and $c$ as

$$A = \begin{bmatrix} A_{\mathcal{B}} & A_{\mathcal{N}} \end{bmatrix}, \quad x = \begin{bmatrix} x_{\mathcal{B}} \\ x_{\mathcal{N}} \end{bmatrix} \quad \text{and} \quad c = \begin{bmatrix} c_{\mathcal{B}} \\ c_{\mathcal{N}} \end{bmatrix}.$$

For simplicity of exposition, we assume here that the $m$-by-$m$ submatrix $A_{\mathcal{B}}$ is nonsingular at every iteration. This assumption is not required in practice as all that is required is the solution of a linear system with coefficient matrix $A_{\mathcal{B}}$.

Upon re-writing the equality constraint in (LP) as $A_{\mathcal{B}}x_{\mathcal{B}} + A_{\mathcal{N}}x_{\mathcal{N}} = b$, we extract $x_{\mathcal{B}} = A_{\mathcal{B}}^{-1}(b - A_{\mathcal{N}}x_{\mathcal{N}})$. In the same manner and using this expression for $x_{\mathcal{B}}$, the objective function becomes $c^T x = c_{\mathcal{B}}^T A_{\mathcal{B}}^{-1} b + (c_{\mathcal{N}} - A_{\mathcal{N}}^T A_{\mathcal{B}}^{-T} c_{\mathcal{B}})^T x_{\mathcal{N}}$. Each iteration of Simplex ensures $x_{\mathcal{B}} \geq 0$. Because Simplex fixes $x_{\mathcal{N}} = 0$, these expressions further simplify to $x_{\mathcal{B}} = A_{\mathcal{B}}^{-1} b$ and $c^T x = c_{\mathcal{B}}^T A_{\mathcal{B}}^{-1} b$. If an improvement is possible, it has to involve increasing a non-basic variable. The last expression of $c^T x$ shows that unit changes in $x_{\mathcal{N}}$ change the objective value at the rate $r = c_{\mathcal{N}} - A_{\mathcal{N}}^T A_{\mathcal{B}}^{-T} c_{\mathcal{B}}$. Therefore, if $r \geq 0$, $(x_{\mathcal{B}}, 0)$ is an optimal solution. The vector $r$ is called the *reduced cost* vector. At each Simplex iteration we look for variables with a negative reduced cost and move one of them—the *entering variable*—into the basis. Since we maintain $m$ basic variables at all times, one variable has to leave the basis—the *leaving variable*. This process of swapping two variables in and out of the basis is called *pivoting*. If we define $\bar{A}_{\mathcal{N}} := A_{\mathcal{B}}^{-1} A_{\mathcal{N}}$ and $\bar{b} := A_{\mathcal{B}}^{-1} b \geq 0$, the choice of the leaving variable is guided by the requirement that the next $x_{\mathcal{B}} = \bar{b} - \bar{A}_{\mathcal{N}} x_{\mathcal{N}} \geq 0$. For a given entering variable $x_j$ ($j \in \mathcal{N}$), we select the leaving variable $x_k$ ($k \in \mathcal{B}$) as that allowing the largest increase possible in the value of $x_j$, i.e., $k \in \operatorname{argmin}_{i \in \mathcal{B}} \{\bar{b}_i / \bar{a}_{ij} \mid \bar{a}_{ij} > 0\}$. From the definition of reduced cost, when $x_j$ enters the basis the objective function improvement is equal to $r_j \bar{b}_k / \bar{a}_{kj}$ where $k$ is the index of the leaving variable. The Simplex method is outlined in Algorithm 4.2.1. There are many pivot rules for choosing the entering and the leaving variables [Terlaky and Zhang, 1993]. Moreover, pivot rules can have a significant impact on the practical performance of the Simplex method.

LP solvers, commercial or free, typically implement several predefined pivot rules. Most of these rules fall into one of the following two categories. The first category is that of rules similar to the original pivot rule presented by Dantzig [1963], where at each iteration a non-basic variable with the smallest negative reduced cost is chosen to enter the basis. Variations on this method choose a variable with a negative reduced cost that is not necessarily the minimum—an approach called *partial pricing*. The second category contains rules based on the steepest-edge method [Goldfarb and Reid, 1977, Harris, 1975, Forrest and Goldfarb, 1992], which work with normalized reduced costs.

**Algorithm 4.2.1** Generic outline of the Simplex method.

**Step 0.** Determine initial $\mathcal{B}$ and $\mathcal{N}$ by identifying a feasible starting point $x = (x_{\mathcal{B}}, 0)$ with $x_{\mathcal{B}} \in \mathbb{R}^m$.

**Step 1.** Compute the reduced cost vector $r = c_{\mathcal{N}} - A_{\mathcal{N}}^T A_{\mathcal{B}}^{-T} c_{\mathcal{B}}$. If $r \geq 0$, the solution is $(x_{\mathcal{B}}, 0)$. Otherwise choose $j$ such that $r_j < 0$ and set $x_j$ as the *entering variable*.

**Step 2.** If $\bar{A}_j \leq 0$ then the problem is unbounded. Otherwise, set $x_k$ as the *leaving variable* where $k \in \text{argmin}_{i \in \mathcal{B}} \left\{ \bar{b}_i / \bar{a}_{ij} \mid \bar{a}_{ij} > 0 \right\}$.

**Step 3.** Pivot: set $\mathcal{N} \leftarrow \mathcal{N} \setminus \{j\} \cup \{k\}$ and $\mathcal{B} \leftarrow \mathcal{B} \setminus \{k\} \cup \{j\}$. Return to Step 1.

It is possible to show that using Dantzig's rule, the iterate moves from the current vertex to an adjacent vertex along an edge $d$ of the feasible polyhedron such that the directional derivative $c^T d$ is as negative as possible. By contrast, a steepest edge rule selects an edge $d$ such that the directional derivative along the normalized edge direction $c^T d / \|d\|$ is as negative as possible. Steepest edge rules are known to outperform Dantzig's original pivot selection by large margins [Wolfe and Cutler, 1963] but require a significantly higher programming effort.

### 4.2.2 Implementing New Pivot Rules

In Step 2 of Algorithm 4.2.1, the definition of $x_{\mathcal{B}}$ implies that $\bar{b} \geq 0$. Therefore, if there exists a $k_0 \in \mathcal{B}$ such that $\bar{b}_{k_0} = 0$ then all choices of $k$ are such that $\bar{b}_k / \bar{a}_{ij} = 0$, and from §4.2.1 we know that performing the pivot will cause no improvements in the objective function. This type of pivot is called a degenerate pivot and an LP for which such pivots occur is said to be degenerate [Greenberg, 1986]. Simplex may be very slow on degenerate LPs or even fail to converge because of cycling. Simple modifications to pivot selection can help avoid cycling, e.g., the method proposed by Bland [1977]. Degeneracy occurs frequently in real-world LPs including but not limited to large-scale set partitioning problems. Raymond et al. [2010] report manpower planning problems with a degeneracy level of 80%, i.e. at each iteration of Simplex we have $\bar{b}_i = 0$ for typically 80% of $i \in \mathcal{B}$. Likewise the patient distribution system (pds) instances of Carolan et al. [1990] have a 80% degeneracy level on average.

In a recent effort to solve large-scale problems with high occurrence of degenerate pivots efficiently, Raymond et al. [2010] introduce the positive edge rule. One of our initial motivations for developing CyLP was to implement the positive edge rule for benchmarking purposes and for application to quadratic and other classes of optimization problems. In the remainder of this section, we explain how new rules may be implemented in CLP and motivate the need for the higher-level mechanism provided by CyLP.

CLP implements both Dantzig's pivot selection and two variants of steepest edge methods with optional partial pricing. In CLP, an execution of the Simplex method on a given problem is abstracted as a C++ class possessing an attribute that represents the pivot selection rule to be used at each iteration. Users specify the pivot rule of their choice by setting this attribute appropriately, the value of the attribute being another C++ class that abstracts the pivot rule itself. New pivot rules may be implemented by subclassing the latter class and overriding certain of its methods. The definition of such a pivot rule in Python can be significantly shorter in terms of number of lines of code and easier in terms of development effort. For example, Dantzig's pivot rule implementation in CLP takes 58 lines of code while a straightforward Python implementation takes only 19 lines—see Listing 4.1. A C++ implementation of the positive edge pivot rule takes 106 lines while we can obtain the same functionality in Python in 38 more readable lines. Conciseness can be crucial in more complex pivot rules. We estimate that the steepest edge method, whose implementation takes about 3800 lines of code in CLP, could be written in less than 500 lines in Python. This makes a Python implementation remarkably easier to develop and debug. Furthermore, since low-level programming details such as memory management are no longer a concern, the programmer can focus almost exclusively on the logic of the pivot rule.

## 4.3 The Positive Edge Rule

The positive edge rule is a recent method to handle degenerate LPs efficiently. To explain this method we consider (LP) and the notation from §4.2.1.

We stated in §4.2.2 that if there is a row $i$ for which $\bar{b}_i = 0$ and $\bar{a}_{ij} > 0$ for some $j$ then we are facing degeneracy. In large-scale LPs it is possible to spend the majority of the solution time performing degenerate pivots.

The positive edge criterion of Raymond et al. [2010] allows us to identify degenerate pivots. Let $Q := A_{\mathcal{B}}^{-1}$ to simplify notation. Define $\mathcal{Z} = \{i = 1, \ldots, m \mid \bar{b}_i = 0\}$ and $\mathcal{P} = \{i = 1, \ldots, m \mid \bar{b}_i > 0\}$. Note that if $\mathcal{Z} = \emptyset$, a nondegenerate Simplex iteration is guaranteed to exist. Accordingly, we partition $Q$ and $A$ row-wise as

$$Q = \begin{bmatrix} Q^{\mathcal{P}} \\ Q^{\mathcal{Z}} \end{bmatrix} \quad \text{and} \quad A = \begin{bmatrix} A^{\mathcal{P}} \\ A^{\mathcal{Z}} \end{bmatrix},$$

where $Q^{\mathcal{P}} \in \mathbb{R}^{|\mathcal{P}|\times m}$, $Q^{\mathcal{Z}} \in \mathbb{R}^{|\mathcal{Z}|\times m}$, $A^{\mathcal{P}} \in \mathbb{R}^{|\mathcal{P}|\times n}$, and $A^{\mathcal{Z}} \in \mathbb{R}^{|\mathcal{Z}|\times n}$. Using a notation similar to that of §4.2.1, we have

$$\begin{bmatrix} \bar{A}^{\mathcal{P}} \\ \bar{A}^{\mathcal{Z}} \end{bmatrix} := \begin{bmatrix} Q^{\mathcal{P}} A \\ Q^{\mathcal{Z}} A \end{bmatrix} = \begin{bmatrix} Q^{\mathcal{P}} A_{\mathcal{B}} & Q^{\mathcal{P}} A_{\mathcal{N}} \\ Q^{\mathcal{Z}} A_{\mathcal{B}} & Q^{\mathcal{Z}} A_{\mathcal{N}} \end{bmatrix} = \begin{bmatrix} I & 0 & Q^{\mathcal{P}} A_{\mathcal{N}} \\ 0 & I & Q^{\mathcal{Z}} A_{\mathcal{N}} \end{bmatrix}, \tag{4.1}$$

where the last equality uses the identity $QA_{\mathcal{B}} = I$.

A variable $x_j$, $j \in \mathcal{B} \cup \mathcal{N}$ is said to be *compatible* if and only if

$$Q^{\mathcal{Z}} A_j = \bar{A}_j^{\mathcal{Z}} = 0.$$

Using the usual identification of $\mathcal{B}$ with $\{1, \ldots, m\}$, we observe from this definition and (4.1) that for $j \in \mathcal{B}$, $x_j$ is compatible if $j \in \mathcal{P}$ and is incompatible if $j \in \mathcal{Z}$. But we are particularly interested in the nonbasic compatible variables because selecting one of them as the entering variable ensures a nondegenerate pivot. From the definition of compatibility we deduce that for a given compatible entering variable $x_j$ ($j \in \mathcal{N}$) and a leaving variable $x_i$ chosen by the ratio test (§4.2.1) the improvement in the objective value, $r_j \bar{b}_i / \bar{a}_{ij}$, is strictly positive, and a non-degenerate pivot is performed. But calculating $\bar{A}_j^{\mathcal{Z}}$ for all variables requires the matrix-matrix product $Q^{\mathcal{Z}} A$. For positive edge to be efficient we must lower the complexity of this identification.

For an arbitrary vector $v \in \mathbb{R}_+^{|\mathcal{Z}|}$, define $w = (Q^{\mathcal{Z}})^T v$. If the variable $x_j$ is compatible, we have

$$w^T A_j = v^T Q^{\mathcal{Z}} A_j = 0. \tag{4.2}$$

Conversely, if $w^T A_j = 0$, can we affirm that $x_j$ is compatible, i.e., $Q^{\mathcal{Z}} A_j = 0$ ? There are two possibilities: either $Q^{\mathcal{Z}} A_j = 0$, in which case $x_j$ is compatible, or $Q^{\mathcal{Z}} A_j \perp v$. For random $v$, the probability of the latter happenning in $\mathbb{R}^{|\mathcal{Z}|}$ is zero. However, in IEEE double precision arithmetic, this probability is proven to be $2^{-62}$ [Raymond et al., 2010]. Therefore, the probability that the statement

$$w^T A_j = 0 \quad \Longrightarrow \quad x_j \text{ is compatible}$$

be erroneous is $2^{-62}$, and this would result in a single degenerate pivot. Since this method does not involve the calculation of updated columns $\bar{A}_j$ its complexity reduces to $O(mn)$—the complexity of the dense matrix-vector product $w^T A$.

The details of the positive edge method are given in Algorithm 4.3.1. In order to obtain a Simplex algorithm equipped with the positive edge rule, Algorithm 4.3.1 should replace Step 1 of Algorithm 4.2.1.

The positive edge method has a parameter that specifies the preferability of compatible variables. We denote this parameter by $0 < \psi < 1$. A value $\psi = 0.4$ means that we prefer a compatible variable over an incompatible one even if the reduced cost of the former is 0.4 that of the latter.

**Algorithm 4.3.1** The Positive Edge Rule

**Step 0.** If $w$ is not initialized or updating $w$ is required, set $\mathcal{P} := \{i \in \mathcal{B} \mid \bar{b}_i > \epsilon\}$ where $\epsilon > 0$ is a prescribed tolerance. Let $v \in \mathbb{R}^m$ be a random vector and fix $v_i = 0$ for all $i \in \mathcal{P}$. Compute $w := A_{\mathcal{B}}^{-T} v$.

**Step 1.** Let $r_{\min} = r_{\text{comp}} = 0$. For each $j \in \mathcal{N}$, do:

1. if $r_j \geq r_{\text{comp}}$, skip to the next variable
2. if $|w^T A_j| < \epsilon$ ($x_j$ is likely compatible) then set $r_{\text{comp}} = r_j$
3. set $r_{\min} = \min(r_{\min}, r_j)$.

**Step 2.** If $r_{\text{comp}} < \psi\, r_{\min}$, choose the compatible variable corresponding to $r_{\text{comp}}$ to enter the basis. Otherwise choose the variable corresponding to $r_{\min}$ and demand an update of $w$ at the next iteration.

At Step 1 of Algorithm 4.3.1, $r_j$ denotes the $j$-th component of the vector of reduced costs defined in §4.2.1, $r_{\text{comp}}$ is the best reduced cost over compatible variables, and $r_{\min}$ is the best overall reduced cost found so far. For more information on the design of the positive edge criterion, we refer the reader to [Raymond et al., 2010].

## 4.4 Implementation Details and Examples

Commercial implementations of Simplex typically allow users to choose a pivot rule among a set of predefined rules, but not to plug in customized or experimental pivot rules. It therefore appears that open-source solvers are the best option if one is to experiment with new pivot rules. One of the leading open-source LP solvers, CLP, is part of the COIN-OR project [Lougee-Heimer, 2003] and has several advantages that make it our solver of choice for the development of CyLP. Firstly, CLP has an object-oriented structure which makes it convenient to extend or modify. Secondly, CLP is written in C++ , a language for which compilers are freely available on most platforms. Finally, the large COIN-OR user base gives confi-

dence that CLP implements the state of the art, and has been exercised and debugged to satisfaction.

In CLP it is possible to define customized pivot rules but a good understanding of its internal structure and of C++ are required. To simplify the exposition we show a partial UML class diagram [Larman, 2001] of CLP in Figure 4.1. A class `ClpSimplex` implements the Simplex method. It has an attribute of type `ClpPrimalColumnPivot`—the base class common to all pivot rules. Every pivot rule must derive from the latter and implement a method called `pivotColumn()` that returns an integer—the index of the entering variable. Figure 4.1 also shows two pivot rules already implemented in CLP.

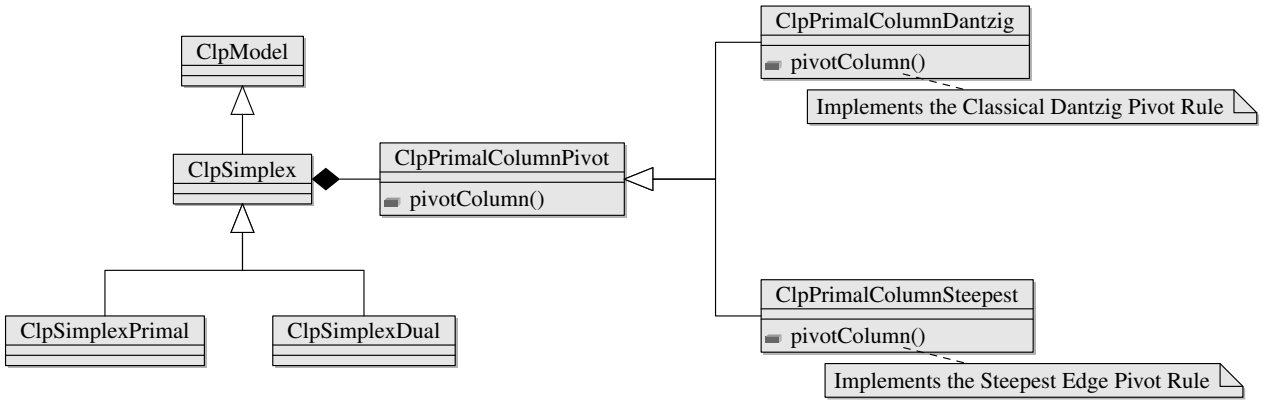



Figure 4.1 Partial UML Class Diagram for CLP

One of the goals of CyLP is to offer users practical and efficient means to implement `pivotColumn()` directly in Python or Cython. Pivot rules need full access to different aspects of a problem, which demands that CyLP wrap a number of components of CLP. In addition, implementing pivot rules often requires defining and maintaining new data structures, vectors and matrices. The standard Python modules Numpy and Scipy [7] facilitate such tasks and make them reasonably efficient. CyLP must therefore be able to interact with those standard packages.

CyLP consists of three layers. The first layer is the auxiliary C++ layer whose role is to enable or facilitate the communication between C++ and Cython. This layer is often required because of technicalities such as the fact that call-by-reference arguments are currently not supported in Cython. Thus, for Cython to communicate with C++ code, it may be necessary to wrap certain functions and methods so as to modify their apparent signature or return value. Consider for example a function that takes an argument by reference, `f(A &a)` where `A`

7. `www.scipy.org`

is a C++ class. One way to work with `f()` in Cython is to interface instead with a C++ wrapper of `f()`: `void f_wrap(A *a) {f(*a);}`. Another example is that of a function that returns an array generated in C++ , say, in the form of a `double*`. Although we can use a `double*` array directly in Cython, we may prefer working with a Numpy array not only because it provides the same constant access time to array elements, but also because it is endowed with a variety of array operations we may need. However, if we require to use the array in Python, the conversion to a Python-understood type like a Numpy array is inevitable. A role of the auxiliary C++ layer is to expose the return array to Cython and Python by wrapping it into a Numpy array data structure. This is achieved by returning the array to Cython as a `PyObject*`, i.e., a pointer to a generic Python object, and subsequently casting this pointer as a pointer to a Numpy array inside Cython. Though it is possible to return a `double*` directly and initialize the Numpy array at the Cython level, we decided against this method for performance reasons.

The second and most important layer is the Cython layer, whose role is to ensure seamless communication between Python and CLP. A collection of Cython files interfaces CLP either directly or indirectly via the auxiliary layer. In the Cython layer, special attention is paid to handling matrices and vectors efficienly while passing them back and forth between C++ and Python.

The third and final layer is the Python layer. This is where we define the callback functions that implement custom pivot rules. These functions will be called from the Cython and/or C++ layers. In a typical use case, we define a pivot rule in Python and pass it over for CLP to use while iterating. The CyLP layers are illustrated in Figure 4.2. The rest of this section is devoted to explaining each of these layers in more detail. To simplify the presentation, we abbreviate `ClpPrimalColumnPivot` to just `Pivot`.

In §4.4.1 we go into some of the details of our implementation of CyLP, the reason being that interfacing a C++ library is not obvious. Moreover, those details may be relevant in other contexts. A reader who wishes to skip over those details may safely go directly to §4.4.2.

### 4.4.1 Implementation Details

As a superset of the Python language, Cython is itself object oriented. If Cython supported inheritance from C++ classes, we could create a Cython subclass of `Pivot` and override `pivotColumn()`. Unfortunately it is currently not possible (at least in an automated way) to inherit from a C++ class inside Cython and we have to find a workaround [8]. What we want is

8. Based on the idea discussed in `tinyurl.com/6sqvd3l`

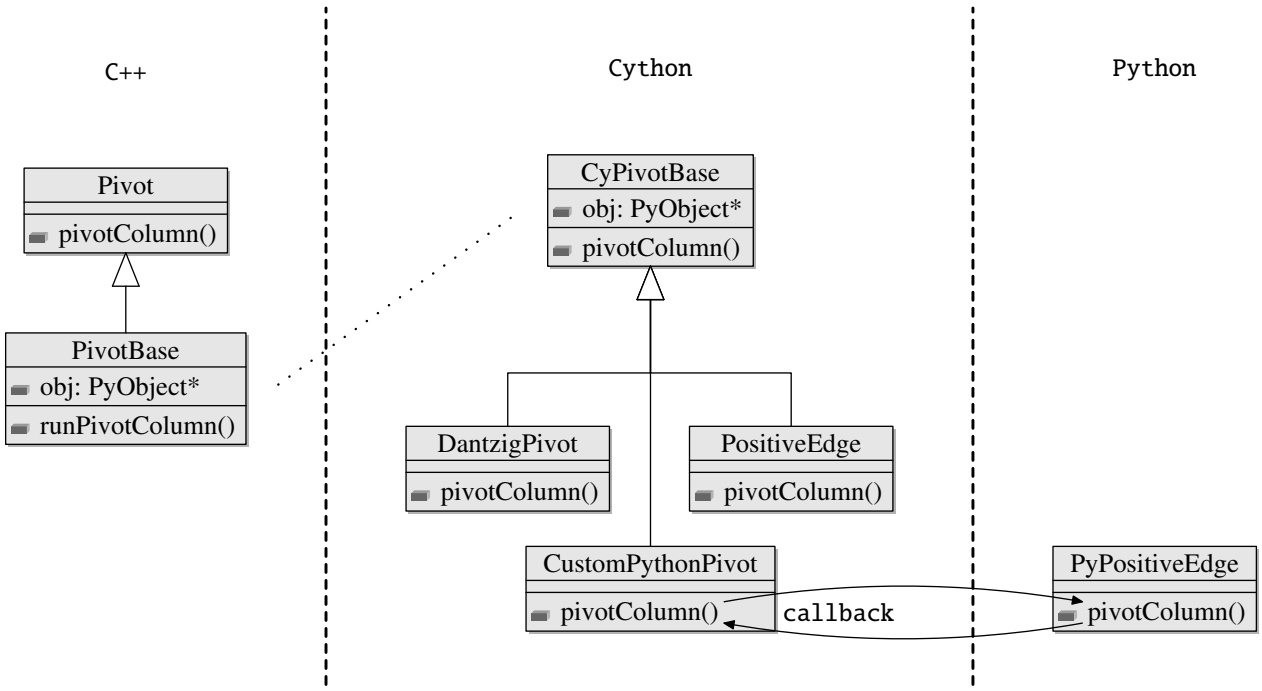



Figure 4.2 Schema of the three layers of CyLP

to have a class equivalent to `Pivot` in Cython, say `CyPivot`, which has the same functionality, i.e., users should be able to define Cython subclasses of `CyPivot` and implement pivot rules by overriding the `pivotColumn()` method. The key to achieving this is understanding the programming concept of *name binding*, which is roughly the process of associating names with objects. The difficulty at the CyLP level can be understood by first considering the following simplification of the definitions in CLP.

```
1  class Pivot{                        // Base class for the user to subclass.
2    public:
3      int pivotColumn() {return -1;} // Generic method for the user to override.
4  };
5
6  class Simplex{
7      public:
8          Pivot *pivotMethod;         // User must bind to a pivot method.
9  };
10
11 class MyFirstPivot: public Pivot{  // User-defined pivot rule.
12   public:
13     int pivotColumn() {return 0;}
14 };
15
16 class MyOtherPivot: public Pivot{  // User-defined pivot rule.
17   public:
18     int pivotColumn() {return 1;}
```

```
};

int main(void) {
    Simplex S;
    S.pivotMethod = new MyFirstPivot();
    printf("%d", S.pivotMethod->pivotColumn());
    S.pivotMethod = new MyOtherPivot();
    printf("%d", S.pivotMethod->pivotColumn());
}
```

The main program above outputs `-1` twice because both times, the `pivotMethod` attribute of `S` is bound to an object of type `Pivot` at compile time, not to objects of type `MyFirstPivot` or `MyOtherPivot`. This behavior is called *static binding*. If we change the signature of the `pivotColumn()` method of the `Pivot` class and declare it `virtual`, binding will be delayed until runtime when the actual type of `pivotMethod` is known, causing the main program to output `0` and `1`. This kind of binding is called *dynamic binding*. This allows CLP to run a `pivotColumn()` of a user-defined pivot rule class.

However, defining a pivot rule class in Cython breaks the inheritance chain—CLP will not be able to recognize our class as a `Pivot` type. As a result, we must bind the C++ `pivotMethod()` to the Cython implementation inside Cython where its type is known.

For instance suppose we define a class `CyDantzigRule` and implement the classic Dantzig pivot rule in its `pivotColumn()` method. To be able to pass a `CyDantzigRule` object to C++ — where `CyDantzigRule` is not recognized as a type—we make use of generic (void) pointers. Suppose we have a void pointer `ptr`, pointing to a `CyDantzigRule` instance. Because casting the void pointer is necessary, Cython requires prior knowledge of the user-defined class, here `CyDantzigRule`, to execute the corresponding `pivotColumn()` method, as illustrated in the following listing:

```
(<CyDantzigRule>(ptr)).pivotColumn()
```

To overcome this limitation we define a Cython class `CyPivot` which shall be the parent of every pivot rule implemented in Cython. Now we are certain that the void pointer can safely be up-cast to a `CyPivot` pointer. The Cython function defined in the following listing

```
cdef int RunPivotColumn(void *ptr):
    return (<CyPivot>(ptr)).pivotColumn()
```

performs this task and runs the `pivotColumn()` method. This is where the call to `pivotColumn()` is bound (dynamically) to `CyDantzigRule.pivotColumn()`.

Taking an additional step ahead, implementing a pivot rule directly in Python would be much more convenient. To this end, we define the `CustomPythonPivot` subclass of `CyPivot`

as we would have done had we wanted to define another pivot rule in Cython. At variance with `CyPivot`, instances of `CustomPythonPivot` have a `pivotMethod` attribute that will be bound to a user-defined Python object implementing `pivotColumn()`. This binding occurs when the user registers their pivot rule for use in the Simplex implementation. This registration stage is illustrated in §4.4.4. For example suppose a user defines the `PyDantzigRule` class and implements a Python version of Dantzig's pivot rule in the `pivotColumn()` method. Upon registering this pivot rule, `CustomPythonPivot.pivotColumn()` is set to call `pivotMethod.pivotColumn()` (instead of implementing a pivot rule itself). In brief, we provide to `CustomPythonPivot` a reference to an actual pivot implementation—a *callback*.

### 4.4.2 Implementation of a Classic Pivot Rule in Cython and Python

Listing 4.1 gives the definition of the classic Dantzig pivot rule in Python. We use the Numpy package to gain performance. We define a class deriving from the `PyPivot` class. In a method that is (and must be) called `pivotColumn()` we first fetch the reduced costs. Then, using the Numpy `where()` function, we gather indices of the variables that are unbounded or those which are at their bounds and whose reduced costs are large enough and have a favorable sign. Among these variables, we choose the one with the maximum absolute value.

```
1  import numpy as np
2
3  class PyDantzig(PyPivot):
4      def pivotColumn(self):
5          s = self.clpModel
6          rc = s.reducedCosts
7          tol = s.dualTolerance()
8
9          indicesToConsider = np.where(s.varNotFlagged & s.varNotFixed &
10                                      s.varNotBasic &
11                                      (((rc > tol) & s.varIsAtUpperBound) |
12                                      ((rc < -tol) & s.varIsAtLowerBound) |
13                                      s.varIsFree))[0]
14
15         abs_rc = abs(rc[indicesToConsider])
16
17         if len(indicesToConsider) > 0:
18             return  indicesToConsider[np.argmax(abs_rc)]
19         return -1
```

Listing 4.1 Dantzig's Pivot Rule in Python

Implementing Dantzig's pivot rule in Cython is essentially similar to the Python implementation in Listing 4.1. The major difference is that in Cython the class is defined as `cdef class CyDantzig(CyPivot)`.

### 4.4.3 Implementation of the Positive Edge Rule

In this section we demonstrate how to implement the positive edge rule of §4.3.

The core of the positive edge pivot rule implementation is essentially similar to that of Dantzig's rule, illustrated in Listing 4.1. The difference is that it should incorporate Algorithm 4.3.1 into its pivot selection process.

We define the Python class `PositiveEdge`, a subclass of `PyPivot`, possessing an `updateW()` method used to perform the optional update of `w` specified in Step 0 of Algorithm 4.3.1, as shown in Listing 4.2. First this method populates `z` with indices of the constraints for which the right-hand-side is smaller in absolute value than `self.EPSILON`—a predefined threshold. Then it sets corresponding elements in `self.w` to a random number. Finally the call to `vectorTimesB_1()` multiplies `self.w` by $A_B^{-1}$ in place.

```
    def updateW(self):
        z = np.where(np.abs(self.rhs) <= self.EPSILON)[0]
        self.w[z] = np.random.random(len(z))
        s = self.clpModel
        s.vectorTimesB_1(self.w)
```

Listing 4.2 Updating $w$

To implement the `pivotColumn()` method of the postive edge rule we use Dantzig's rule implementation in Listing 4.1 as a starting point.

Each variable compatibility check requires a dot product, i.e. $A_j^T w$ for $j \in \mathcal{N}$. Instead, we choose to check the compatibility of all the non-basic variables at once by performing a vector by matrix multiplication, i.e. $A_{\mathcal{N}}^T w$, which is more efficient. To this end, we use a wrapper of `ClpModel`'s `transposeTimesSubset()` using the call

```
s.transposeTimesSubset(idx, w, Aw)
```

which multiplies $w$ by the rows of $A_{\mathcal{N}}^T$ specified in the list `idx`. The result is stored in `Aw`, which is a Numpy array. To get the indices of compatible variables we can use Numpy's function `where()` once again:

```
compVars = idx[np.where(abs(Aw[idx]) < self.EPSILON)[0]]
```

We next identify a compatible variable with maximum reduced cost:

```
compRc = abs(rc[compVars])          #Reduced costs of the compatible variables
maxCompIdx = compVars[np.argmax(compRc)]
```

We compare `rc[maxCompIdx]` with the reduced cost of the variable chosen by Dantzig's method considering the preferability of compatible variables, by the predefined parameter

$\psi$, and decide either to choose a compatible or an incompatible variable. If an incompatible variable is chosen, we call `updateW()` to reconstruct $w$ before moving on to the next iteration.

As a result, we are able to implement the positive edge method in Python in 38 lines while the same implementation in C++ takes 106 lines.

### 4.4.4 A Complete Example Usage of CyLP

Suppose that we have a LP defined in an MPS file `lp.mps`. To solve this problem in Python using the positive edge pivot method we use:

```
s = CyClpSimplex()
s.readMps("lp.mps")
s.preSolve(tol=1.0e-8)  # Optional presolve step.
pivot = PositiveEdgePivot(s)
s.setPivotMethod(pivot)
s.primal()              # Executes primal Simplex.
```

Listing 4.3 Using Custom Pivot Rules

where we first create an instance of `CyClpSimplex`—a class which interfaces CLP's `ClpSimplex`. After reading the problem from `lp.mps`, we create an instance of `PositiveEdge` and register it with `s`. Then we solve the model using the CLP's primal Simplex method.

### 4.4.5 Modeling Facilities

As an alternative to reading from a file, CyLP provides intuitive modeling facilities to express linear programming problems. Listing 4.4 shows how to model (4.3), solve it using primal Simplex, add a new constraint and solve again.

$$
\begin{array}{lrcrcrcrcrcr}
\text{minimize} & & & x_0 & - & 2x_1 & + & 3x_2 & + & 2y_0 & + & 2y_1 \\
\text{s.t.} & & & x_0 & + & 2x_1 & & & & & \leq & 5 \\
 & & & x_0 & & & + & x_2 & & & \leq & 2.5 \\
 & 2 & \leq & x_0 & & & & & + & y_0 + 2y_1 & \leq & 4.2 \\
 & 2 & \leq & & & & & x_2 & & y_1 & \leq & 3 \\
 & & & & & & & & & (y_0, y_1) & \geq & 0 \\
 & & & & & & & & & 1.1 \leq x_1 & \leq & 2 \\
 & & & & & & & & & 1.1 \leq x_2 & \leq & 3.5.
\end{array}
\tag{4.3}
$$

```
import numpy as np
from CyLP.cy import CyClpSimplex
```

```
3  from CyLP.py.modeling.CyLPModel import CyLPArray
4
5  s = CyClpSimplex()
6
7  x = s.addVariable('x', 3)
8  y = s.addVariable('y', 2)
9
10 # Define coefficient matrices and bound vectors.
11 A = np.matrix([[1., 2., 0],[1., 0, 1.]])
12 B = np.matrix([[1., 0, 0], [0, 0, 1.]])
13 D = np.matrix([[1., 2.],[0, 1]])
14 a = CyLPArray([5, 2.5])
15 b = CyLPArray([4.2, 3])
16 u = CyLPArray([2., 3.5])
17
18 # Add constraints and bounds to model.
19 s += A * x <= a
20 s += 2 <= B * x + D * y <= b
21 s += y >= 0
22 s += 1.1 <= x[1:3] <= u
23
24 # Define the objective function
25 c = CyLPArray([1., -2., 3.])
26 s.objective = c * x + 2 * y.sum()
27
28 s.primal()              # Solve. Solution: [0.2  2.   1.1  0.   0.9]
29
30 s += x[2] + y[1] >= 2.1 # Add a cut.
31 s.primal()              # Warm start. Solution: [ 0.  2.  1.1  0.   1.]
```

Listing 4.4 Creating and solving an LP using CyLP modeling facility.

The `addVariable()` method returns a `CyLPVar` object which we use later to add constraints and bounds. Lines 11–16 define coefficient matrices and vectors. Vectors are defined using `CyLPArray` objects instead of the more familiar Numpy arrays, whereas matrices may be defined using Numpy's `matrix` objects. The reason lies in the way Numpy array operators take precedence. If we compare a `CyLPVar` object, `x` with a Numpy array `b` using the expression `b >= x` the `>=` operator of `b` is called and returns a Numpy array with the same dimension as `b` and all its elements set to `False`, which is of no significance since `b` is trying to compare itself with another object that it does not know of. We would expect, instead, the `<=` operator of `x` to execute, and to save `b` as the lower bound on `x`. To this end, we use `CyLPArray` objects which are Numpy arrays in most respects except that they concedes performing an operation if the other operand is a `CyLPVar` object.

Constraints and variable bounds are declared using the `addConstraint()` method or the in-place addition operator as in lines 19–22. The objective function is defined by setting `CyClpSimplex`'s `objective` attribute. Once the LP is defined, the `writeMps()` method allows

us to write the problem to file in `mps` format. This makes the choice of modeling with CyLP independent of the choice of solver. In CyLP, LPs may be solved using primal or dual Simplex. The modeling tool also makes it easy to generate a problem dynamically, as in cut-generation or branch-and-cut techniques in integer programming, by allowing users to add constraints at any time and re-solve the problem with a *warm start*, i.e., using the solution of the last execution as the initial point for the new problem. In the next section we demonstrate how CyLP can be used to solve MIPs and how users can customize the branch-and-cut process using Python callbacks.

### 4.4.6 Mixed Integer Programming with CyLP

Much in the same way as CyLP enables to customize the pivot selection rule in CLP, it also enables to customize the solution process of MIPs using a branch-and-cut strategy. This is made possible by interfacing COIN-OR's CBC library. Specifically, custom cuts may be input by the user by way of COIN-OR's CGL Cut-Generation Library [9]. CGL supplies generators for a collection of well-known cuts, such as Gomory, clique and knapsack cover cuts. CyLP facilitates access to these cut generators. Although the interface is still at a preliminary stage, it already offers a certain level of flexibility which, when combined with user-designed pivot rules, may produce powerful variants of the solution process.

In this section, we illustrate how to add integrality restriction to variables, how to solve a MIP and tune the solution process. Finally, we explain how CyLP can be used to write Python callbacks to customize the branch-and-cut process.

To mark variables as integers we use the `setInteger()` method. In the LP of Listing 4.4, for example, $x_1$ and $x_2$ can be marked as integers by adding `s.setInteger(x[1:3])`. Once the MIP is modeled, we solve it using an instance of the `CyCbcModel` class.

Considering the case of reading the problem from a file, Listing 4.5 illustrates how to solve a MIP using two cut generators. To obtain the `CyLPModel` of the problem, instead of `readMps()`, we read the LP using `extractCyLPModel()`. We use the returned model to access the problem's variables in line 8, and mark a subset of them as integers in line 9. Calling `getCbcModel()` solves the initial relaxation using CLP and returns an instance of the `CyCbcModel` class, which is an interface to CBC's `CbcModel` class, that implements a branch-and-cut procedure. We then add two cut generators—one generating Gomory cuts of at most 100 variables and the other generating knapsack cover cuts. Afterwards, we solve the problem and print out the solution.

9. `projects.coin-or.org/Cgl`

```
s = CyClpSimplex()

# Read problem from file. To have access to the CyLPModel of
# the problem we use extractCyLPModel method instead of readMps.
model = s.extractCyLPModel('lp.mps')

# Mark variables x5 to x9 as integers
x = model.getVarByName('x')
s.setInteger(x[5:10])

# Solve initial relaxation and obtain a CyCbcModel object.
cbcModel = s.getCbcModel()

# Create a Gomory cut generator and a Knapsack cover cut generator.
gomory   = CyCglGomory(limit=100)
knapsack = CyCglKnapsackCover(maxInKnapsack=50)

# Add cut generators to CyCbcModel.
cbcModel.addCutGenerator(gomory,   name="Gomory")
cbcModel.addCutGenerator(knapsack, name="Knapsack")

cbcModel.branchAndBound()  # Solve.
print cbcModel.primalVariableSolution
```

Listing 4.5 Integer Programming with CyLP

CBC provides the capability to customize its branch-and-cut node selection process by writing C++ callbacks. For this purpose, CyLP enables us to, instead, use Python. Suppose that we wish to implement a simplistic approach of traversing the branch-and-cut tree. The strategy is to look for nodes with the least number of unsatisfied integrality constraints. In case of a tie, at first we select the deepest node (i.e., a *depth-first* strategy). But whenever an integer solution is found we break the tie by choosing the highest node (*i.e., a breadth-first* strategy). Listing 4.6 illustrates how to implement this strategy by defining a class that inherits from the relevant base class `NodeCompareBase`.

Our subclass must implement `compare()` to determine the preference in node selection, `newSolution()`, which will be run by CBC after a solution is found to perform a possible change of strategy, and `every1000Nodes()`, which is similar to `newSolution()` but is called after CBC has visited 1000 nodes. To use `SimpleNodeCompare` in Listing 4.6, we set the node comparison method by registering an instance `snc` of `SimpleNodeCompare` with the `CbcModel` object using `cbcModel.setNodeCompare(snc)`.

## 4.5 Numerical Experiments

In this section, we first examine the performance hit caused by implementing a pivot rule in Python or Cython as opposed to C++ . We choose Dantzig's pivot selection rule because it is simple enough to ensure a fair comparison across different implementations. Later, we

```
class SimpleNodeCompare(NodeCompareBase):
    def __init__(self):
        self.method = 'depth'   # Default strategy.

    def compare(self, x, y):
        "Return True if node y is better than node x."
        if x.nViolated != y.nViolated:
            return (x.nViolated > y.nViolated)
        if x.depth == y.depth:
            return x.breakTie(y)   # Break ties consistently.
        if self.method == 'depth':
            return (x.depth < y.depth)
        return (x.depth > y.depth)

    def newSolution(self, model, objContinuous, nInfeasContinuous):
        "Cbc calls this after a solution is found in a node."
        self.method = 'breadth'

    def every1000Nodes(self, model, nNodes):
        "Cbc calls this every 1000 nodes for possible change of strategy."
        return False        # Do not demand a tree re-sort.
```

Listing 4.6 A simple node comparison impelementation in Python

demonstrate how CyLP can be used to examine the effectiveness of the positive edge pivot rule.

We choose the Netlib LP benchmark [10] for the first part, which contains 93 LPs of diverse dimensions and sparsity. All our experiments are conducted on computers with Intel Xeon 2.4GHz CPUs and 49GB of total shared memory. Let $t^d_{cpp}$, $t^d_{cy}$ and $t^d_{py}$ denote the execution times of Algorithm 4.2.1 using C++ , Cython and Python versions of Dantzig's pivot rule, respectively.

Figure 4.3 show the performance profile [Dolan and Moré, 2002] of the execution times. The figure illustrates that, as expected, the C++ implementation is always faster but that the Cython and Python versions are essentially equivalent to one another. It also demonstrates that for 50% of the instances, the Cython and Python versions are less than 3 times slower than the C++ version.

We next select those Netlib instances that take more than 5 seconds to solve using the C++ implementation of Dantzig's rule and measure the performance hit caused by using Cython and Python by computing the *slowdown* factors $t^d_{cy}/t^d_{cpp}$ and $t^d_{py}/t^d_{cpp}$. The results are given in the form of a bar chart in Figure 4.4. Problems are sorted by C++ execution time from `greenbeb`, taking 5 seconds, to `dfl001`, taking 9729 seconds. The average slowdown is 2.3 and in the most difficult instance, `dfl001`, is about 1.4. Our observation is that as problems become moderately large, the performance gap shrinks to a point where it no longer has a significant impact.

10. `www.netlib.org/lp/data`

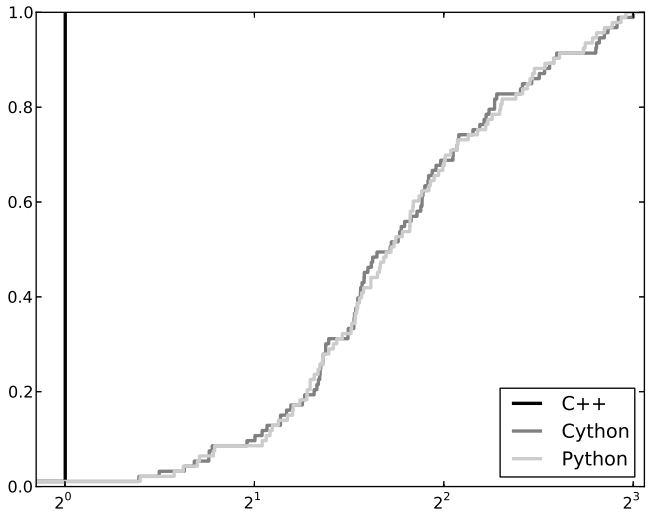



Figure 4.3 Performance profile for the execution time of the primal Simplex Algorithm using the C++ , Cython and Python implementations of Dantzig's pivot rule.

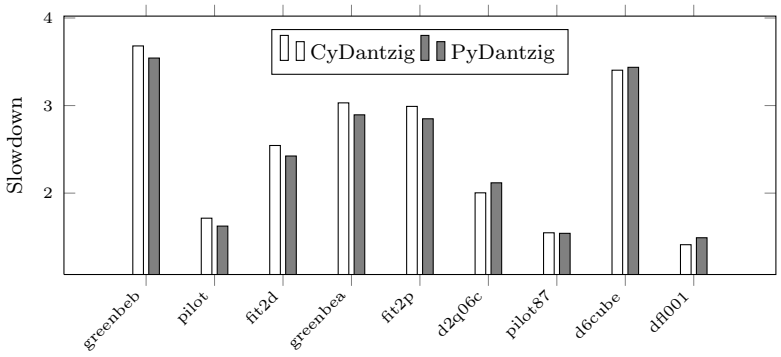



Figure 4.4 Performance hit caused by Python and Cython compared to C++

For the second part of the numerical tests we choose from among the `pds` instances from Mittlemann's benchmark [Carolan et al., 1990], which are large, sparse and highly degenerate—good target problems for the positive edge method.

Let $t^p_{cpp}$ and $t^p_{py}$ be the execution times of the positive edge method using C++ and Python implementations, respectively. We compute $t^d_{cpp}/t^p_{cpp}$ and $t^d_{py}/t^p_{py}$ which indicate the speedups gained by using the positive edge rule relative to Dantzig's rule in C++ and Python. Results of these tests appear in Table 4.1. In this table, $n$ and $m$ denote the number of variables and constraints of the instance, respectively, $i_d$ and $i_p$ are the numbers of iterations necessary to solve the instance using Dantzig's pivot rule and the positive edge rule, respectively. Python reports higher speedups than C++ but the iteration reductions are identical. The reason is that positive edge is adding an almost equal overhead to each iteration in C++ and Python. For example in `pds-06`, the average C++ iteration time is 0.0002 seconds for Dantzig and 0.0004 seconds for positive edge. As for Python, the average iteration time is 0.0019 seconds for Dantzig and 0.0017 seconds for positive edge. This means that positive edge is adding an extra 0.0002 seconds on average to each Dantzig iteration which is relatively more costly for C++ . This also shows that a careful implementation of the positive edge rule in Python runs at almost the same speed as in C++ , each iteration taking $3 \times 10^{-6}$ seconds longer.

Table 4.1 demonstrates the superiority of the positive edge method over Dantzig's pivot rule for larger `pds` instances. In fact, increasing speedups for both implementations show that the effectiveness of the positive edge rule increases as the problem size grows, both in terms of run time and in terms of number of pivots.

Table 4.1 Speedup of the positive edge method relative to Dantzig's rule.

| Instance | $n$ | $m$ | $i_d$ | $i_p$ | C++ speedup | Python speedup |
|---|---|---|---|---|---|---|
| pds-02 | 7535 | 2953 | 553 | 583 | 0.32 | 0.78 |
| pds-06 | 28655 | 9881 | 7759 | 2816 | 1.03 | 2.85 |
| pds-10 | 48763 | 16558 | 37939 | 6890 | 2.50 | 6.12 |
| pds-20 | 105728 | 33874 | 293668 | 31584 | 3.92 | 9.78 |
| pds-30 | 154998 | 49944 | 597447 | 65657 | 4.20 | 9.75 |
| pds-40 | 212859 | 66844 | 1504587 | 139376 | 5.20 | 10.85 |

### 4.6 Discussion and Future Work

CyLP, available from `github.com/mpy/CyLP`, provides a high-level scripting framework to define and customize aspects of the solution process of LPs and MIPs using COIN-OR's CLP and CBC. It uses callback methods to let users define new pivot rules in Python and have CLP use them during primal Simplex. We demonstrated this feature by implementing the positive edge pivot rule in C++ and Python. We feel that the ease of programming and flexibility offered by implementing pivot rules in Python outweigh the slowdown caused by using a high-level interpreted programming language. Moreover, this slowdown becomes minor as problem size grows.

Besides the pivot selection rules discussed throughout this paper, we also implemented the Last In First Out and the Most Often Selected rules described by Terlaky and Zhang [1993], each in less than 30 lines of code. However, those rules also demand to restrict the leaving variable selection, which is not currently possible in CyLP. The reason is that in CLP, entering variable selection is designed to be customized by users and is defined in separate classes whereas a leaving variables rule is built into CLP's `ClpSimplexPrimal` class. Nevertheless, future improvements to CyLP will remove this limitation.

Currently, custom pivot rules may only be passed to the primal Simplex solver. In the future, we wish to provide facilities to implement dual pivot rules in Python. As an improvement to the integer programming facilities—where we are already capable of defining the branch-and-cut node comparison rule in Python—we will consider adding the capability to script cut generators in the same manner.

In follow-up research, we consider Wolfe's pivot rule to solve the KKT system of a convex quadratic program [Wolfe, 1959]. The KKT system of a QP is a set of linear equations if we set aside the complementarity conditions. Wolfe proposes to solve an LP to find a feasible point for this system by using a specific pivot strategy to take care of complementarity. Our goal in doing so will be to investigate the application of the positive edge rule and of the constraint aggregation techniques of Elhallaoui et al. [2010] to convex quadratic programming. The interface to CLP described in the present paper will let us implement Wolfe's rule and construct modified linear programs easily. We hope other users find CyLP equally valuable in their research.

Cython is a powerful intermediate language to enable interaction between low-level high-performance libraries and Python. We expect that other types of optimization solvers would benefit from similar scripting capabilities. In nonconvex optimization, the flexibility and power of solvers such as IPOPT [Wächter and Biegler, 2006] would, in our opinion, be greatly

enhanced were users able to plug in their own linear system solver or barrier parameter update using Python.

## REFERENCES

A. Aides. Cython wrapper for IPOPT. `http://code.google.com/p/cyipopt`. [Online; accessed 2-November-2011].

M. Berkelaar. lpsolve, A Mixed Integer Linear Programming Software. `http://lpsolve.sourceforge.net`. [Online; accessed 2-November-2011].

R. G. Bland. New finite pivoting rules for the Simplex method. *Mathematics of Operations Research*, 2(2):103–107, 1977. ISSN 0364765X.

W. Carolan, J. Hill, J. Kennington, S. Niemi, and S. Wichmann. Empirical Evaluation of the KORBX Algorithms for Military Airlift Applications. *Operations Research*, 38:240–248, 1990.

B. A. Cipra. The best of the 20th century: editors name top 10 algorithms. *SIAM News*, 33 (4):1–2, 2000.

CLP. COIN-OR Linear Programming. `https://projects.COIN-OR.org/Clp`. [Online; accessed 2-November-2011].

G. B. Dantzig. *Linear programming and extensions*. Princeton University Press, New Jersey, 1963.

E. Dolan and J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming B*, 91:201–213, 2002.

J. Dongarra and F. Sullivan. Guest editors' introduction: The top 10 algorithms. *Computing in Science and Engineering*, 2:22–23, 2000. ISSN 1521-9615. DOI: 10.1109/MCISE.2000.814652.

I. Elhallaoui, A. Metrane, , G. Desaulniers, and F. Soumis. An improved primal simplex algorithm for degenerate linear programs. *INFORMS Journal on Computing*, 2010. DOI: 10.1287/ijoc.1100.0425.

J. J. Forrest and D. Goldfarb. Steepest-edge Simplex algorithms for linear programming. *Mathematical Programming*, 57:341–374, 1992. ISSN 0025-5610. DOI: 10.1007/BF01581089.

D. Goldfarb and J. K. Reid. A practicable steepest-edge Simplex algorithm. *Mathematical Programming*, 12:361–371, 1977. ISSN 0025-5610. DOI: 10.1007/BF01593804.

H. J. Greenberg. An analysis of degeneracy. *Naval Research Logistics Quarterly*, 33:635–655, 1986.

P. M. J. Harris. Pivot selection methods of the Devex LP code. In R. W. Cottle, L. C. W. Dixon, B. Korte, M. J. Todd, E. L. Allgower, W. H. Cunningham, J. E. Dennis, B. C. Eaves, R. Fletcher, D. Goldfarb, J.-B. Hiriart-Urruty, M. Iri, R. G. Jeroslow, D. S. Johnson, C. Lemarechal, L. Lovasz, L. McLinden, M. J. D. Powell, W. R. Pulleyblank, A. H. G. Rinnooy Kan, K. Ritter, R. W. H. Sargent, D. F. Shanno, L. E. Trotter, H. Tuy, R. J. B. Wets, E. M. L. Beale, G. B. Dantzig, L. V. Kantorovich, T. C. Koopmans, A. W. Tucker, P. Wolfe, M. L. Balinski, and Eli Hellerman, editors, *Computational Practice in Mathematical Programming*, volume 4 of *Mathematical Programming Studies*, pages 30–57. Springer Berlin Heidelberg, 1975. ISBN 978-3-642-00766-8. DOI: 10.1007/BFb0120710.

K. L. Hoffman and M. Padberg. Solving airline crew scheduling problems by branch-and-cut. *Management Science*, 39:675–682, June 1993. DOI: 10.1287/mnsc.39.6.657.

H. Koepke. Cython wrapper for CPLEX. `http://www.stat.washington.edu/~hoytak/code/pycpx/index.html`, a. [Online; accessed 2-November-2011].

H. Koepke. Cython wrapper for lpsolve. `http://www.stat.washington.edu/~hoytak/code/pylpsolve/index.html`, b. [Online; accessed 2-November-2011].

C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, 2nd Edition*. Prentice Hall, 2001.

R. Lougee-Heimer. The common optimization interface for operations research. *IBM Journal of Research and Development*, 47(1):57–66, 2003. DOI: 10.1147/rd.471.0057. URL `www.COIN-OR.org`.

A. Makhorin. GLPK, GNU Linear Programming Kit. `http://www.gnu.org/s/glpk`. [Online; accessed 2-November-2011].

M. Padberg and G. Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Rev.*, 33:60–100, February 1991. ISSN 0036-1445. DOI: 10.1137/1033004.

PuLP. An LP modeler written in Python. `http://code.google.com/p/pulp-or`. [Online; accessed 2-November-2011].

V. Raymond, F. Soumis, A. Metrane, and J. Desrosiers. Positive edge: A pricing criterion for the identification of non-degenerate Simplex pivots. Cahier du GERAD G-2010-61, GERAD, Montréal, Québec, Canada, 2010.

P. J. S. Silva. Pycoin, interface to some COIN packages. `http://www.ime.usp.br/~pjssilva/software.html`, 2005. [Online; accessed 2-November-2011].

T. Terlaky and S. Zhang. Pivot rules for linear programming: A survey on recent theoretical developments. *Annals of Operations Research*, 46-47:203–233, 1993. ISSN 0254-5330. DOI: 10.1007/BF02096264.

A. Wächter and L. T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106:25–57, 2006. DOI: 10.1007/s10107-004-0559-y. URL `https://projects.COIN-OR.org/Ipopt`.

P. Wolfe. The Simplex method for quadratic programming. *Econometrica*, 27(3):382–398, 1959.

P. Wolfe and L. Cutler. Experiments in linear programming. In Graves and Wolfe, editors, *Recent Advances in Mathematical Programming*. McGraw-Hill, New York, 1963.

# CHAPTER 5

# EXTENSIONS OF POSITIVE EDGE IN QUADRATIC PROGRAMMING

## 5.1 Introduction

Degeneracy arises in quadratic programs (QP) as it does in linear programs (LP) ; when we have dependent equality constraints or the strict complementarity fails for the inequality constraints. Unless equipped with appropriate measures, QP solvers may struggle when dealing with degeneracy, which can cause infinite loops or critical slowdowns in the solution process.

QP solution techniques can be divided into two distinct categories, active-set methods and interior-point methods. In this chapter we exclusively consider solving degenerate QPs by means of active-set-based methods. Gould et al. [2011] provide an insight into QP degeneracy from an interior-point perspective.

The similar nature of QP's active-set methods and LP's Simplex method, in that they both move from one point to another on the faces of the feasible region, makes it natural to generalize degeneracy techniques for LP to QP. For example, the well-known Bland's rule [Bland, 1977], which guarantees termination of the Simplex method in a finite number of iterations, is extended by Chang and Cottle [1980] to QPs. In a comparable pattern, a technique to resolve degeneracy by Fletcher [1988] is adapted for QPs by Fletcher [1993]. Following the same paradigm, in this chapter we inspect the extension of the positive edge rule [Raymond et al., 2010] to solve degenerate QPs.

The positive edge method defines a pivot selection rule for the Simplex algorithm that is proved to be effective especially on large-scale degenerate LPs. We incorporate positive edge into two Simplex-like QP methods: one by Wolfe [1959], which involves slight modifications to normal Simplex pivot rules, and a reduced gradient approach [Bazaraa et al., 2006].

This chapter is organized as follows. Background on the quadratic Simplex, degeneracy, and the positive edge pivot rule appears in §5.2. In §5.3, we equip Wolfe's QP technique with the positive edge rule, in what we refer to as the WP method. For the implementation, we require an LP solver that allows the definition of pivot methods. We choose a software called CyLP , which is built upon CLP, and provides a framework to implement pivot methods in the Python programming language (Chapter 4).

Afterwards, in §5.4, we integrate positive edge into the reduced gradient method, available in CLP. Finally in §5.5, we investigate a scaling technique that aims to "reduce the nonlinearity" of general QPs.

### Notation

Throughout this chapter we denote matrices and vectors by capital and small letters respectively. For index sets we use capital letters in a calligraphic font, e.g. $\mathcal{B}$ and $\mathcal{N}$. For a matrix $M$, $M_{\mathcal{B}}$ is the submatrix of $M$ containing those columns indexed in $\mathcal{B}$. In the same manner we use superscripts, e.g., $M^{\mathcal{B}}$ to denote row submatrices. The $j$-th column of $M$ is $M_j$, whose $i$-th element is denoted by $m_{ij}$.

## 5.2 Background

### 5.2.1 Quadratic Simplex

We consider solving convex quadratic programs (QP) of the form

$$\underset{x\in\mathbb{R}^n}{\text{minimize}} \quad c^T x + \tfrac{1}{2}x^T G x \quad \text{s.t.} \qquad Ax = b, x \geq 0. \tag{QP}$$

where $c \in \mathbb{R}^n$, $G \in \mathbb{R}^{n\times n}$ is symmetric and positive semi-definite, $A \in \mathbb{R}^{m\times n}$ and $b \in \mathbb{R}^m$.

The optimality conditions of (QP) can be written as

$$Gx - A^T y - z = -c \tag{5.1a}$$
$$Ax = b \tag{5.1b}$$
$$(x, z) \geq 0 \tag{5.1c}$$
$$z^T x = 0. \tag{5.1d}$$

In the QP literature, the name "quadratic simplex" vaguely refers to a wide range of active-set methods that resemble the Simplex method. During this project, we experimented with two methods in this category, namely Wolfe's method [Wolfe, 1959], and a reduced gradient method [Bazaraa et al., 2006, §10.8]. We describe these approaches in the following sections.

### 5.2.2 Degeneracy

A QP is called degenerate if it has linearly dependent equality constraints, or if strict complementarity fails at a given point. We say that strict complementarity fails at $x_0 \in \mathbb{R}^n$ if there exists an $i$ for which $x_{0_i} = z_{0_i} = 0$ where $(x_0, z_0)$ satisfy (5.1). Figure 5.1 shows two examples of degeneracy in QPs. In both figures, $x^*$ is the unconstrained minimizer, and lines $c_1$ to $c_4$ correspond to linear inequality constraints with signs chosen such that $x^*$ is feasible. In the left plot, $x^*$ satisfies the constraint as equality but removing the constraint from the problem will not affect the solution. In the right plot, at $x_0$, only constraints $c_1$ and $c_2$ are enough to specify the feasible region, but all four constraints are active at that point, making it difficult for an active-set method to guess the right active set—one that permits non-zero steps towards the solution ; in this example $\{c_1\}$. Degeneracy is common and has a significant effect on the performance of QP solvers. Gould et al. [2011] gather degeneracy-related statistics about test problems of two QP benchmarks by Maros and Meszaros [1999] and Gould et al. [2003].

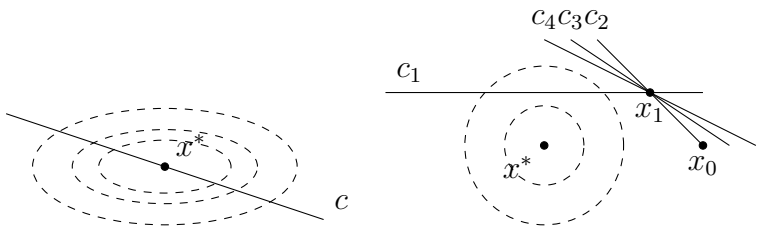



Figure 5.1 Degeneracy in quadratic programming

### 5.2.3 Positive Edge

Dealing with degeneracy in LPs typically involves either perturbation, e.g., lexicographic rule [Dantzig et al., 1955], or a modification to the Simplex pivot selection rule, e.g., Bland's rule [Bland, 1977]. Contributing to the latter type, Raymond et al. [2010] propose the *positive edge* rule—a method that detects degenerate pivots at each Simplex iteration with a reasonable complexity. In the rest of this section we first briefly explain Simplex [Dantzig, 1963]. Afterwards we describe the positive edge method in detail.

Consider an LP of the form

$$\underset{x\in\mathbb{R}^n}{\text{minimize}}\ c^T x \quad \text{s.t.} \qquad Ax = b,\ x \geq 0, \tag{LP}$$

where $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$, $A \in \mathbb{R}^{m\times n}$. Simplex fixes $n - m$ variables to zero and calls them *nonbasic* and considers the rest of the variables as *basic*. At each iteration, it swaps basic and nonbasic variables, in what is called *pivoting*, until it obtains a solution.

Denote the index set of the basic variables by $\mathcal{B}$, and that of the non-basic variables by $\mathcal{N}$. Substituting $x$ by $(x_\mathcal{B}, x_\mathcal{N})$ in $Ax = b$ and multiplying both sides by $A_\mathcal{B}^{-1}$ we can express the basic variables in terms of the non-basic variables as $x_\mathcal{B} = A_\mathcal{B}^{-1}(b - A_\mathcal{N} x_\mathcal{N})$. Replacing this in the objective function we have $c^T x = c_\mathcal{B}^T A_\mathcal{B}^{-1} b + (c_\mathcal{N} - A_\mathcal{N}^T A_\mathcal{B}^{-T} c_\mathcal{B})^T x_\mathcal{N}$. Consequently, we can predict the changes in the objective function based on the changes of the nonbasic variables, i.e. each unit change of variable $x_j, j \in \mathcal{N}$ alters the value of the objective function exactly $r_j = c_j - A_j^T A_\mathcal{B}^{-T} c_\mathcal{B}$ units—which is called the *reduced cost* of $x_j$. Therefore, At each iteration of Simplex, we search for a variable with a negative reduced cost, say $x_j$, to reduce the objective function. However, increasing the value of $x_j$ is restricted by the positivity constraint $x \geq 0$. We raise the value of $x_j$ until a basic variable, say $x_k$, vanishes. More specifically, we choose $x_k$ as the *leaving variable* where $k = \text{argmin}_{i\in\mathcal{B}}\{\bar{b}_i/\bar{a}_{ij} \mid \bar{a}_{ij} > 0\}$ where $\bar{A} = A_\mathcal{B}^{-1} A$ and $\bar{b} = A_\mathcal{B}^{-1} b$. By replacing $x_j$ in the basis, we have $x_j = \bar{b}_k/\bar{a}_{kj}$ and thus the objective function improvement of $r_j \bar{b}_k/\bar{a}_{kj}$. Therefore, if there exists $i \in \mathcal{B}$ where $\bar{b}_i = 0$ and $\bar{a}_{ij} > 0$ then the iteration will not change the objective function value, and so it is called a *degenerate pivot*. In the light of this, Positive Edge avoids degenerate pivots by choosing the entering variable, $x_j$, from the set $\{x_j \mid \forall i \in \{1, 2, ..., m\}$ if $\bar{b}_i = 0$ then $\bar{a}_{ij} = 0\}$, which leads to the definition of *compatible* variables.

Let $\mathcal{Z} := \{i = 1, \ldots, m \mid \bar{b}_i = 0\}$ and $\mathcal{P} := \{i = 1, \ldots, m \mid \bar{b}_i > 0\}$.

**Definition 3.** *Variable $x_j, j \in \{1, 2, ..., n\}$ is compatible if and only if $\bar{A}_j^\mathcal{Z} = 0$.*

Partitioning $\bar{A}$ by its rows and defining $Q := A_\mathcal{B}^{-1}$ we have

$$\bar{A} = \begin{bmatrix} \bar{A}^\mathcal{P} \\ \bar{A}^\mathcal{Z} \end{bmatrix} = \begin{bmatrix} Q^\mathcal{P} A \\ Q^\mathcal{Z} A \end{bmatrix} = \begin{bmatrix} Q^\mathcal{P} A_\mathcal{B} & Q^\mathcal{P} A_\mathcal{N} \\ Q^\mathcal{Z} A_\mathcal{B} & Q^\mathcal{Z} A_\mathcal{N} \end{bmatrix} = \begin{bmatrix} I & 0 & Q^\mathcal{P} A_\mathcal{N} \\ 0 & I & Q^\mathcal{Z} A_\mathcal{N} \end{bmatrix}. \tag{5.2}$$

We infer from the compatibility definition and (5.2) that $x_j$ is compatible for $j \in \mathcal{P}$ and incompatible for $j \in \mathcal{Z}$. But for the entering variable choice, we must search for nonbasic compatible variables, $x_j, j \in \mathcal{N}$. The cost of finding all these variables is to perform a matrix-by-matrix operation, $Q^\mathcal{Z} A_\mathcal{N}$, at each iteration of Simplex, which is too expensive.

To lower the complexity of the compatible variable search, we define a random vector $v \in \mathbb{R}_+^{|\mathcal{Z}|}$ and let $w^T := v^T Q^{\mathcal{Z}}$. If $x_j$ is compatible, then we have $w^T A_j = v^T Q^{\mathcal{Z}} A_j = 0$. But is it true that if $w^T A_j = 0$ then $x_j$ is compatible and $Q^{\mathcal{Z}} A_j = 0$ ? If $w^T A_j = 0$ then we must have either $Q^{\mathcal{Z}} A_j = 0$ or $Q^{\mathcal{Z}} A_j \perp v^T$. Raymond et al. [2010] prove that the probability of the latter to happen in double precision is $2^{-62}$ if $v$ is populated by a bitwise Bernoulli random generator explained in detail in the mentioned article. Therefore, if $w^T A_j = 0$ then with a probability of $1 - 2^{-62}$ we have $Q^{\mathcal{Z}} A_j = 0$. In a situation where this statement becomes untrue we might detect an incompatible variable as compatible and perform a single degenerate iteration.

### 5.2.4 CyLP

In the course of implementing WP, we must develop and combine two pivot methods: Wolfe and positive edge. Therefore, we require a Simplex LP solver that allows user-defined pivot rules. Commercial LP solvers often come with several well-known predefined pivots, e.g. the classic Dantzig pivot [Dantzig, 1963], the steepest edge rule [Wolfe and Cutler, 1963, Goldfarb and Reid, 1977, Forrest and Goldfarb, 1992], and the DEVEX rule [Harris, 1975]. But in our case we need an open-source application, such as CLP [1], which permits defining pivot rules. COIN-OR's CLP has an object-oriented design which makes it a good option for modifications of any sort, including pivot rule definition. But being written in C++ , we have to be familiar with low-level programming techniques in order to develop and debug the code. As a result, we use the framework provided by CyLP that allows us to define the CLP's pivot rule in the high-level programming environment of Python. We explain CyLP in more details in Chapter 4.

## 5.3 Step One: Positive Edge and Wolfe's Method

### 5.3.1 Wolfe's Method

Wolfe proposes to solve (5.1) by solving an LP comprising (5.1a), (5.1b) and (5.1c) as constraints:

$$\begin{aligned} \underset{x,y,z,a_1,a_2}{\text{minimize}} \quad & \mathbf{1}_n^T a_1 + \mathbf{1}_m^T a_2 \\ \text{s.t.} \quad & Gx - A^T y - z + a_1 = -c \\ & Ax + a_2 = b \end{aligned}$$

1. `https://projects.coin-or.org/Clp`

$$(x, z, a_1, a_2) \geq 0,$$

where $\mathbf{1}_k \in \mathbb{R}^k$ is a vector with all of its elements equal to 1. By modifying the normal Simplex pivot rule we can also take (5.1d), the *complementarity conditions*, into account. The rule is simple: A variable can enter the basis only if its complement is not in the basis or its complement will leave the basis in the same iteration.

Wolfe proposes to first solve an LP,

$$\begin{aligned} \underset{a_1}{\text{minimize}} \quad & e_1^T a_1 \\ \text{s.t.} \quad & Gx - A^T y - z + s_1 - s_2 = -c \\ & Ax + a_1 = b \\ & (x, z, a_1, s_1, s_2) \geq 0 \\ & y = z = 0, \end{aligned} \tag{W1}$$

where $a_1 \in \mathbb{R}^m$ is the vector of artificial variables and $s_1 \in \mathbb{R}^n$ and $s_2 \in \mathbb{R}^n$ are slack and surplus variables to relax the dual-feasibility condition. The solution of (W1) is a feasible point satisfying $Ax = b$ only if the corresponding objective function value is zero. To ensure that this solution also satisfies the complementarity conditions we set $z = 0$, and to simplify further we also set $y = 0$, by removing $z$ and $y$ from (W1). Then we solve the following problem using the optimal basis of (W1) as the initial basis.

$$\begin{aligned} \underset{s_1, s_2, x, y, z}{\text{minimize}} \quad & e_1^T s_1 + e_2^T s_2 \\ \text{s.t.} \quad & Gx - A^T y - z + s_1 - s_2 = -c \\ & Ax = b \\ & (x, z, s_1, s_2) \geq 0 \end{aligned} \tag{W2}$$

The missing complementarity conditions, $x^T z = 0$, are satisfied at every Simplex iteration by using Wolfe's pivot rule. Wolfe [1959] proves that for strictly convex QPs, i.e. when $G$ is positive definite, this procedure ends in finite number of iterations, a maximum of $\binom{3n}{n}$. However, when $G$ is positive semi-definite, one needs to apply regularization to ensure finite termination.

### 5.3.2 Algorithm

Implementation of WP requires modifying Dantzig's classic entering variable selection to Algorithm 5.3.1. Subscripts $a$, $c$, and $e$ are used to signify, respectively, the variable with the best reduced cost, the compatible variable with the best reduced cost, and the current pivot's entering variable.

**Algorithm 5.3.1** The Positive Edge Rule Combined with Wolfe's method

**Step 0.** Find a variable, $x_a$, with the best reduced cost, $r_a$.

**Step 1.** Find a compatible variable, $x_c$, with the best reduced cost among the compatible variables, $r_c$.

**Step 2.** Choose the entering variable, $x_e$, to be $x_c$, if $\psi r_a > r_c$, and $x_a$ otherwise, where $0 < \psi < 1$ determines how strongly we favor compatible variables.

**Step 3.** If the complement of $x_e$ is in the basis and will not leave the basis at the same iteration, i.e. it will not be chosen by the ratio test, we reject $x_e$, and ban it from being chosen again before a successful iteration is performed. Go to Step 0. Otherwise declare $x_e$ as the entering variable.

In Step 0 we use Dantzig's rule to find a variable with the best reduced cost. Then, in Step 1, we examine compatible variables with negative reduced costs. In Step 2 we decide whether or not we prefer a compatible variable over an incompatible one, even if its reduced cost is worse, determined by the value of a predefined parameter, $0 < \psi < 1$. For example, $\psi = 0.1$ means that we prefer a compatible variable if its reduced cost is at least a tenth of the best reduced cost. Finally in Step 3, we check if proceeding with the selected entering variable will cause violation of the complementarity constraints, i.e. its complement is in the basis and the ratio test will not choose the complement as the leaving variable in the same iteration, in which case we reject the entering variable. This process is repeated until optimality is reached. Since Wolfe's method's termination does not depend on the order in which we choose entering variables, and all that positive edge does is to give higher priority to compatible variables as entering variable candidates, WP terminates in a finite number of iterations following the same convergence results as Wolfe's original approach.

In the following section, we examine a general QP structure which is unfavorable to WP's performance.

**A Case where WP Fails**

In Algorithm 5.3.1, Step 3, we risk rejecting compatible variables suggested by the positive edge rule. This contradiction becomes a critical disadvantage of the method for particular problems. To examine further, consider solving a QP of the form

$$\underset{x\in\mathbb{R}^n}{\text{minimize}} \quad c^Tx + \tfrac{1}{2}x^Tx \quad \text{s.t.} \qquad x \geq 0, \tag{5.3}$$

using Wolfe's method. We readily have a solution to (W1), $x = z = 0$. On the other hand, problem (W2) becomes

$$\begin{aligned} \underset{s_1,s_2,x,z}{\text{minimize}} \quad & e_1^Ts_1 + e_2^Ts_2 \\ \text{s.t.} \qquad & x - z + s_1 - s_2 = -c \\ & (x, z, s_1, s_2) \geq 0, \end{aligned}$$

where $z^Tx = 0$. Notice that here, the constraint matrix $A \in \mathbb{R}^{n\times 4n}$ is equal to

$$\begin{bmatrix} I_n & -I_n & I_n & -I_n \end{bmatrix}$$

We claim that variable $x_j$ is compatible if and only if $z_j$ is compatible. To prove this, we use the same notation as in §5.2.3. In addition, for a given variable $x_j$,we denote its corresponding column in $A$ and $\bar{A}$ by $A_{x_j}$ and $\bar{A}_{x_j}$ respectively. For every variable $x_j$ and its complement $z_j$, we have

$$\bar{A}_{x_j}^{\mathcal{Z}} = A_{\mathcal{B}}^{-1}I_j = A_{\mathcal{B}}^{-1}(-a_{z_j}^{\mathcal{Z}}) = -\bar{A}_{z_j}^{\mathcal{Z}}. \tag{5.4}$$

Therefore $\bar{a}_{x_j}^{\mathcal{Z}} = 0$ if and only if $\bar{a}_{z_j}^{\mathcal{Z}} = 0$, hence the result follows.

Let $\mathcal{B}_x \subset \mathcal{B}$ and $\mathcal{B}_z \subset \mathcal{B}$ index the basic $x$'s and $z$'s respectively. A non-basic variable $x_k$ is compatible only if $k \in \mathcal{B}_z$. Similarly, a non-basic variable $z_l$ is compatible only if $l \in \mathcal{B}_x$. From the definition of Wolfe's algorithm, we know that such compatible pivots will be rejected unless the entering variable replaces its complement in the basis at the same iteration—which seldom happens in our observations.

On the other hand, the objective function makes $s_1$ and $s_2$ unfavorable to enter the basis. Therefore, in a typical Simplex iteration, there exists no compatible variable that passes Wolfe's test. In practice, in this particular kind of problem, Wolfe's method rejects close to 100% of the compatible variables.

To solve (5.3) using WP, a workaround is to restate problem (5.3) equivalently as

$$\underset{x\in\mathbb{R}^n}{\text{minimize}} \quad c^Tx + \tfrac{1}{2}x^Tx \quad \text{s.t.} \quad x - w = 0,\ w \geq 0. \tag{5.5}$$

The solution to (W1) is $(x, w) = (0, 0)$. We start from this point to solve (W2) which is to

$$\begin{array}{llccccccccccccc}
\underset{s_1,s_2,x,z,k}{\text{minimize}} & & & & & & e_1^Ts_1 & + & e_2^Ts_2 & + & e_3^Ts_3 & + & e_4^Ts_4 & & \\
\text{s.t.} & x & - & Iy & & + & s_1 & - & s_2 & & & & & = & -c \\
 & & + & Iy & - & z & & & & + & s_3 & - & s_4 & = & 0 \\
 & x & & & & & & & & & & & - \ w & = & 0 \\
 & & & & & & & & & & & & (x, z, s_1, s_2, s_3, s_4) & \geq & 0 \ ,
\end{array}$$

where $z^Tw = 0$ and $s_3 \in \mathbb{R}^n$ and $s_4 \in \mathbb{R}^n$ have the same role as $s_1$ and $s_2$. Notice that the compatible variables, $z$ and $w$, do not appear in the same constraint. Therefore, Equation (5.4) does not hold for $\bar{A}^{\mathcal{Z}}_{z_j}$ and $\bar{A}^{\mathcal{Z}}_{w_j}$, where $j$ is an arbitrary index. As a result, by solving (5.5) instead of (5.3), we avoid the high rate of compatible-variable rejection situation explained above. Of course, this is from a theoretical point of view, whereas in practice, (5.5) has $n$ more constraints than (5.3) which has a significant negative effect on the performance.

### 5.3.3 Implementing Wolfe with Positive Edge

In this section we first explain the use of CyLP to implement pivot rules and some implementation details specific to WP. Later we describe how we model (W1) and (W2) using CyLP to solve QPs.

Towhidi and Orban [2012] provide description of how to implement pivot rules in CyLP. First, we create a class derived from `PivotPythonBase`. Then we implement a call-back method called `pivotColumn()` that returns the index of the entering variable. However, this procedure is not enough for Wolfe's method because our decision on whether or not to accept an entering variable could be delayed until we find the corresponding leaving variable. To consider this, we implement another callback method called `isPivotAcceptable()` which CyLP calls at each iteration just before the actual pivot occurs, giving the user a last chance to reject a pivot. In the case of Wolfe's method, the implementation of `isPivotAcceptable()` is shown in Listing 5.1.

```
    def isPivotAcceptable(self):
        s = self.clpModel
        cl = self.complementarityList

        ...
```

```
        # if the complement is basic and not leaving the basis
        if s.getVarStatus(cl[enteringIndex]) == 1 and \
            cl[enteringIndex] != leavingVarIndex:
            # Mark the variable before rejection
            self.Banned[enteringIndex] = True
            return False

        # The pivot is accepted, reset the marked variables
        self.Banned = np.array(self.dim * [False], np.bool)
        return True
```

Listing 5.1 Accepting or rejecting a pivot in Wolfe's pivot rule

Wolfe's QP solution method consists of two stages. First we create and solve an LP, (W1), the solution of which is a feasible point for (QP). In the second stage, we add the dual variables, $y$ and $z$, along with their constraints, to the problem to obtain (W2). We solve (W2) using the special pivot method explained earlier in this section to guarantee the satisfaction of complementarity conditions. We demonstrate these two steps more clearly in Listing 5.2. Notice that `WolfePivotPE` is the class implementing the WP method.

```
s = CyLPSimplex()
x = s.addVariable('x', nVars)

# Modeling (W1), Primal feasibility constraints
m += A * x == b
s += x >= 0

# solving (W1)
s.primal()

# Modeling (W2)
# Adding dual variables
z = s.addVariable('z', nVars)
y = s.addVariable('y', nEquality)
sp = s.addVariable('sp', nVars)
sm = s.addVariable('sm', nVars)

# Dual feasibility constraints
s += G * x - A.T * y - z + sp - sm == -c

# Positivity
s += z >= 0
s += sp >= 0
s += sm >= 0

# Objective function
s.objective = sp.sum() + sm.sum()

# Changing the pivot rule, setting complementarity constraints for x, z
p = WolfePivotPE(s)
p.setComplement(s.cyLPModel, x, z)
```

```
s.setPivotMethod(p)

s.primal()  # Solving (W2)
```

Listing 5.2 Solving a QP using Wolfe's method

### 5.3.4 Numerical Experiments

In this section we compare our implementations of the Wolfe and WP methods.

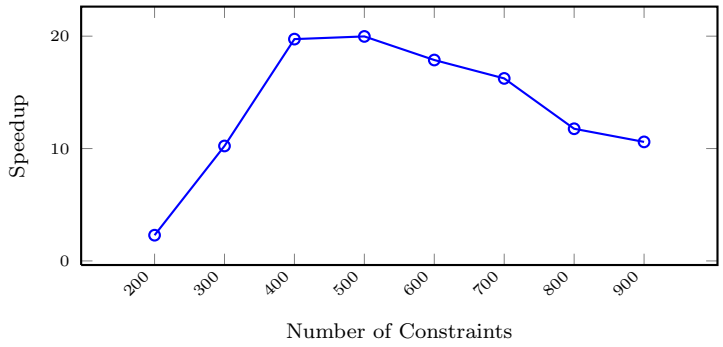
First, we explain how we choose appropriate test problems. In linear problems, positive edge is efficient on problems having certain properties, namely, when they are highly sparse and degenerate. In QP, we have additional constraints on the problems structure for positive edge to be efficient, one of which was discussed in §5.3.2. Therefore, we create a random QP generator that generates problems meeting these requirements.

The QP generator creates problems with set-covering-like constraints and a convex quadratic objective function. It also has a parameter to control the number of non-zero elements in each column. For the tests that we present here, we generated problems with 500 and 800 variables, with Hessian equal to the identity matrix, and no bounds on their variables. They only differ by the number of constraints, $m$, and the number of non-zero elements per column, which we set to $m/10$.

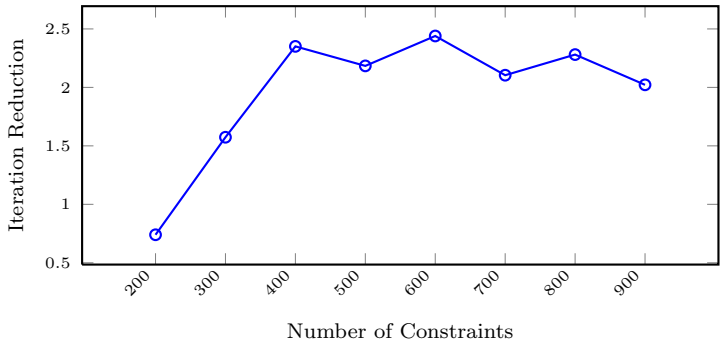
If we denote the execution time of Wolfe and WP by $t_w$ and $t_{wp}$, the *speedup* is defined as $t_w/t_{wp}$. In the same manner, we define *iteration reduction* to be $i_w/i_{wp}$ where $i_w$ and $i_{wp}$ are the number of iterations of Wolfe and WP methods respectively.

In Figure 5.2(a), we demonstrate the speedup gained by WP with respect to Wolfe's method in problems with 500 variables. It is clear that WP is particularly more efficient on problems with close to 500 constraints. The reason is that in (W2) we have both $A$ and $A^T$ in the constraints, therefore each variables or constraint in QP contributes to the number of constraints is (W2). Figure 5.2(c) which exhibits the speedup of WP in problems with 800 variables also confirms WP's relatively higher efficiency on square problems.

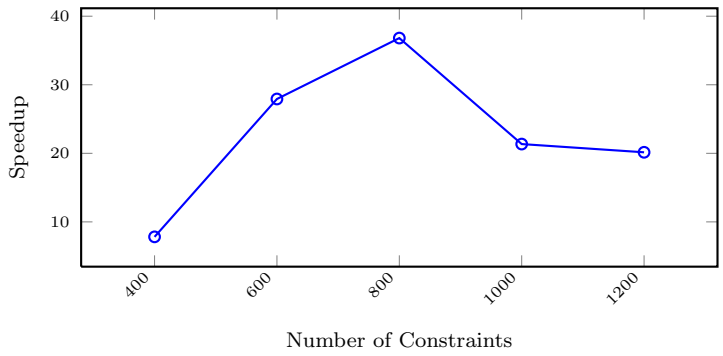
In Figure 5.2(a) the maximum speedup is close to 20, while in Figure 5.2(c) we are able to obtain about 40 showing that the performance of WP becomes more distinguishable on larger problems with more variables.




(a) Speedup in QPs with 500 variables




(b) Iteration reduction in QPs with 500 variables




(c) Speedup in QPs with 800 variables

Figure 5.2 The effect of Wolfe+PE on specially generated QPs

## 5.4 Step Two: Positive Edge and the Reduced Gradient Method

### Reduced Gradient Method

In the optimality conditions of (QP), equality (5.1a) can be rewritten as

$$r = Gx + c - A^T y \geq 0,$$

where $r$ is called the *reduced gradient*. Note that in the special where $G = 0$, an LP, the reduced gradient becomes the reduced cost. This motivates us to find a solution to (5.1) using a simplex-like approach. To find an improving direction at a given feasible point, we look for increasing the values of variables $x_j$ where $r_j < 0$. Depending on the number of nonbasic variables that we allow to change at the same time we obtain different variations of the method. The *reduced gradient* method [Bazaraa et al., 2006, §10.6] allows all nonbasic variables to change at once, which often results in small steps to maintain primal feasibility. On the other hand, the *convex-simplex method of Zangwill* [Bazaraa et al., 2006, §10.7] lets only one nonbasic variable become positive, similarly to the Simplex method, and usually requires more iterations to solve the problem. In a trade-off, more efficient variants allow a subset of nonbasic variables to change at a single iteration [Bazaraa et al., 2006, §10.8]. Contrary to Wolfe's method, here, we solve an LP with the same constraints as QP, $Ax = b, x \geq 0$.

Since a solution to QP may have more than $m$ positive values, we use the notion of *super basic*, variables indexed by $\mathcal{S}$, which is a subset of nonbasic variables that are strictly positive.

Let $\mathcal{B}$ and $\mathcal{N}$ index basic and nonbasic variables, where $A_\mathcal{B} \in \mathbb{R}^{m\times m}$ is nonsingular, and for $i \in \mathcal{B} \cup \mathcal{N}$ we have $x_i \geq 0$. Notice that we are not forcing nonbasic variables to zero.

At iteration $k$, we are interested in moving from the current point $x^k$ to another point $x^{k+1}$ such that $f(x^{k+1}) \leq f(x^k)$, where $f(x) = c^T x + \frac{1}{2}x^T Gx$. We define the direction as $d := x^{k+1} - x^k$. For $d$ to be a feasible direction, we must have $Ad = A(x^{k+1} - x^k) = b - b = 0$, so $Ad = A_\mathcal{B} d_\mathcal{B} + A_\mathcal{N} d_\mathcal{N} = 0$, and therefore

$$d_\mathcal{B} = -A_\mathcal{B}^{-1} A_\mathcal{N} d_\mathcal{N}. \tag{5.6}$$

For $d$ to be a descent direction, we must have $\nabla f(x^k)^T d < 0$. Using (5.6) we obtain

$$\begin{aligned}\nabla f(x^k)^T d &= \nabla_{x_\mathcal{B}} f(x^k)^T d_\mathcal{B} + \nabla_{x_\mathcal{N}} f(x^k)^T d_\mathcal{N} \\ &= \left[\nabla_{x_\mathcal{N}} f(x^k)^T - \nabla_{x_\mathcal{B}} f(x^k)^T A_\mathcal{B}^{-1} A_\mathcal{N}\right] d_\mathcal{N}\end{aligned}$$

$$= r_{\mathcal{N}}^T d_{\mathcal{N}}. \tag{5.7}$$

where $r_{\mathcal{N}}^T := \left[\nabla_{x_{\mathcal{N}}} f(x^k)^T - \nabla_{x_{\mathcal{B}}} f(x^k)^T A_{\mathcal{B}}^{-1} A_{\mathcal{N}}\right]$ is called the *reduced gradient*.

Therefore, for $d$ to be a descent direction we must have $r_{\mathcal{N}}^T d_{\mathcal{N}} < 0$. In addition, we demand that $d_j \geq 0$ if $x_j = 0$ to avoid zero steps. An evident choice for $d_{\mathcal{N}}$ is to define its elements, $d_j, j \in \mathcal{N}$ as

$$d_j := \begin{cases} -r_j & \text{if } r_j \leq 0 \\ -x_j r_j & \text{if } r_j > 0. \end{cases} \tag{5.8}$$

This definition prevents performing short steps by downscaling $d_j$ in cases where $r_j > 0$ and $x_j$ is a small positive number. If $d = 0$, then $x^k$ is a KKT point and we stop. Otherwise, we compute a step length by solving a line search problem,

$$\begin{aligned} \underset{\lambda}{\text{minimize}} \quad & f(x^k + \lambda d) \\ \text{s.t.} \quad & 0 \leq \lambda \leq \lambda_{max}, \end{aligned}$$

where

$$\lambda_{max} := \begin{cases} \min\limits_j \{\frac{-x_j^k}{d_j}\} & \text{if } d \ngeq 0 \\ \infty & \text{if } d \geq 0. \end{cases}$$

By repeating this procedure, $x^k$ converges to a KKT point [Bazaraa et al., 2006, Theorem 10.6.3].

More efficient variants of the reduced gradient method, allow only a subset of nonbasic variables, which are called *super basic* variables, to have nonzero values and fix others to zero. As a result, the direction vector $d$ has fewer nonzero entries which makes it more likely to have larger steps. At each iteration, we make maximum improvement using only the basic and super basic variables. Then we look for a nonbasic variable $x_j$ with the smallest $r_j$. We change its status to super basic and we repeat the procedure.

### 5.4.1 Implementation

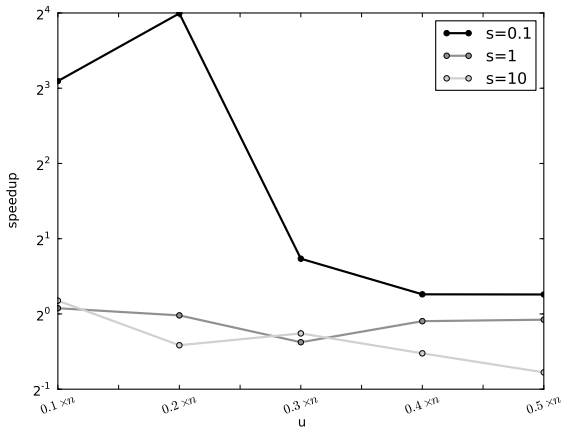
To test positive edge combined with a reduced gradient method we use CLP, as it contains a QP Simplex. The class implementing this method is called `ClpSimplexNonlinear`. We modify method `DirectionVector()`, where a nonbasic variable with the smallest reduced gradient is chosen to become super basic, to consider compatibility of a variable as a decision factor. Therefore, in addition to searching for a variable, say $x_k$, with the smallest reduced gradients, $r_k$, we look for a compatible variable, e.g. $x_c$, with the smallest reduced gradient, $r_c$. If $r_c < \psi r_k$, where $0 < \psi < 1$, we choose $x_c$, otherwise we select $x_k$. As before, the parameter $\psi$ controls the preferability of a compatible variable over an incompatible one. Notice that this rule does not affect the convergence as we always look among all the variables for negative reduced gradients.

In the numerical tests, we refer to this implementation as CLP+PE.

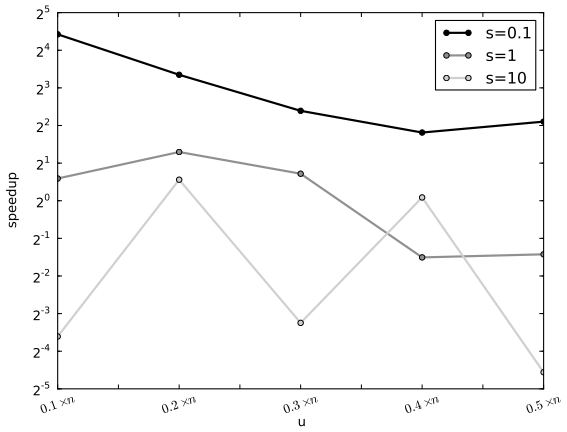
### 5.4.2 Numerical Results

The fact that in reduced gradient methods we solve a system with $m$ constraints— compared to the $n + m$ constraints of Wolfe's method— allows us to experiment with QPs of larger dimension. To generate test problems, we choose the degenerate LPs of the PDS instances, and add a quadratic term to their objective function. This is inspired by scheduling problems in which we assign tasks to workers or machines, while we require that the schedules of different workers be fairly balanced, i.e. even though the main objective is to assign every task, we prefer a schedule in which a worker is not assigned many more tasks than the others. We may introduce quadratic terms to the objective function to minimize the variation between schedules.

To test the effect of a quadratic term in the objective function, we simply add diagonal matrices as the Hessian of the objective function, i.e. $H_{ii} = s$ for $i \in \{0, 1, ..., u\}$ where $s$ and $u$ are parameters. We choose $s \in \{0.1, 1, 10\}$ and $u \in \{0.1n, 0.2n, 0.3n, 0.4n, 0.5n\}$, where $n$ is the number of variables and the values of $u$ are rounded to the closest integer. We solve the problems once with CLP's reduced gradient method and compute the execution time $t_{clp}$. Then, we solve the problem using CLP+PE and call the execution time $t_{pe}$. We define the speedup to be $t_{clp}/t_{pe}$. Figure 5.3 shows the result of these tests on `pds-10` and `pds-20`. We observe that lower densities of the Hessian lead to better performance of CLP+PE. Moreover, with the smallest tested $s$, 0.1, we obtain significant speedups for smaller values of $u$. Also, with $s = 0.1$, the speedup is always greater than or equal to 1.




(a) pds-10




(b) pds-20

Figure 5.3 CLP vs CLP+PE on modified PDS instances

The observation that weaker quadratic terms cause better performance in CLP+PE, motivates a natural scaling technique that we explain in the next section.

## 5.5 Step Three: Scaling

We know that positive edge improves the performance of the primal Simplex method on linear problems. On the other hand, on QPs, attempts to apply positive edge on Simplex-like methods are not consistently conclusive. Here, we present an idea to scale a QP so that it becomes more "linear", i.e. according to a chosen norm, the quadratic part is less significant than the linear term.

We define $\tilde{x} := \alpha x$ and substitute it in QP to obtain

$$\underset{\tilde{x}\in\mathbb{R}^n}{\text{minimize}} \quad \tilde{c}^T\tilde{x} + \tfrac{1}{2}\tilde{x}^T\tilde{G}\tilde{x} \quad \text{s.t.} \qquad \tilde{A}\tilde{x} = b, \tilde{x} \geq 0. \tag{5.9}$$

where $\tilde{c} = \frac{1}{\alpha}c$, $\tilde{G} = \frac{1}{\alpha^2}G$, and $\tilde{A} = \frac{1}{\alpha}A$. If we find a solution to (5.9), $\tilde{x}^*$, then $x^* = \frac{1}{\alpha}\tilde{x}^*$ is a solution to (QP). Nevertheless, notice that in (5.9), the linear term is multiplied by $\frac{1}{\alpha}$ while the quadratic part is multiplied by $\frac{1}{\alpha^2}$. Therefore, a careful choice of $\alpha$ could make quadratic part less significant relative to the linear term, and thus presumably more favorable to positive edge.

We consider $l^2$-norm to measure the vector of linear coefficients and the Frobenius norm for the Hessian matrix. Now, for example, if we want $\|\tilde{G}\|_F/\|\tilde{c}\| = 1/10$, we choose $\alpha := 10\|G\|_F/\|c\|$. In the next section, we present numerical results to observe the effect that different values of $\alpha$ have on the solution process.

### 5.5.1 Numerical Results

We test scaling QPs by running CLP+PE on the original and the scaled version of instances of Maros and Meszaros [1999]. Figure 5.4 demonstrates the result in the form of a color map. The vertical axis represents the instances sorted by their ratio of $\|G\|_F/\|c\|$, which varies from about 0.001 to $6.7 \times 10^6$. The horizontal axis shows 12 different settings for $\alpha$, the first corresponding to the original problem, and the next 11 to scaled problems with increasing values of $\alpha$. We use different shades of gray to show different speedups we obtain relative to CLP's original reduced gradient method, invoked in CLP by using the primal simplex option on a QP. The white color corresponds to instances with less than 1 speedup or, in other

words, a slowdown. Darker shades of gray signify higher speedups, and finally the black color is for speedups between 3 and 5000.

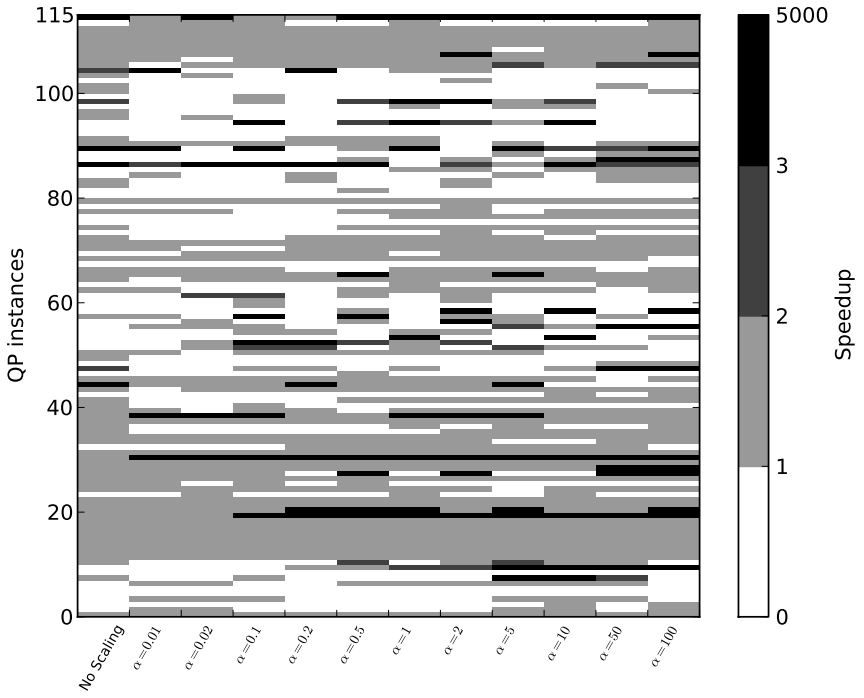



Figure 5.4 CLP+PE Speedup compared to CLP for different values of $\alpha$

In Figure 5.4 we observe many gray cells which prove the positive effect of scaling over positive edge given the right value for $\alpha$. On the other hand, no obvious pattern emerges from the figure. The figure gives no knowledge as to how to choose an $\alpha$ for a problem with a specific $\|G\|_F/\|c\|$ so that positive edge would be effective.

Nevertheless, if we consider individual problems, we find interesting instances. We select some of these problems in Table 5.1. The column corresponding problem `DTOC3`, shows a clear advantage of using large values of $\alpha$, as we see a monotonic increase in speedup. Setting $\alpha = 100$ results in CLP+PE to run a remarkable 4703 times faster than CLP's original primal simplex QP solver—from 993 seconds down to 0.21 seconds. However, in other instances, although we obtained extraordinary speedups, the choice of $\alpha$ is not as obvious.

Table 5.1 The effect of scaling on positive edge speedup over selected QP instances

| | speedup | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\alpha$ | AUG2D | AUG3DC | DTOC3 | DUAL1 | GOULDQP2 | LISWET3 | LISWET6 | MOSARQP1 | MOSARQP2 | PRIMALC8 | QPCSTAIR | VALUES |
| 0.01 | 0.81 | 0.88 | 0.7 | 0.76 | 242.82 | 1.55 | 0.6 | 0 | 0 | 1.48 | 9.04 | 1 |
| 0.02 | 0.75 | 0.88 | 0.53 | 1 | 166.94 | 1.46 | 0.61 | 0.22 | 0 | 1 | 0.13 | 1 |
| 0.1 | 0.98 | 0.93 | 0.85 | 1.45 | 242.82 | 1.38 | 0.36 | 1.09 | 0 | 1.19 | 3.51 | 1.31 |
| 0.2 | 0.98 | 0.59 | 1.31 | 1.45 | 445.17 | 0.63 | 0.51 | 1.23 | 0.22 | 0.67 | 0.7 | 1 |
| 0.5 | 1.05 | 1.08 | 1.6 | 1.45 | 445.17 | 0.52 | 1.25 | 0.67 | 1.24 | 31 | 1.48 | 1 |
| 1 | 1.09 | 1.08 | 2.31 | 1.45 | 2671 | 0.69 | 0.69 | 1.32 | 0.01 | 0.51 | 5.27 | 1 |
| 2 | 1.15 | 1.16 | 2.82 | 1.45 | 2671 | 0.55 | 102.55 | 0.65 | 1.29 | 31 | 0.46 | 1 |
| 5 | 1.14 | 1.08 | 1308.45 | 2.67 | 2671 | 2.64 | 0.11 | 0.99 | 0.03 | 0.01 | 6.31 | 1.31 |
| 10 | 1.19 | 1.08 | 1400.34 | 1.45 | 2671 | 1.11 | 4.72 | 1.22 | 1.6 | 0.08 | 2 | 1 |
| 50 | 1.12 | 1.27 | 1440.81 | 2.67 | 242.82 | 4.61 | 0.41 | 3.1 | 34.33 | 5.17 | 2.21 | 3.5 |
| 100 | 1.48 | 1.39 | 4703.03 | 2.67 | 242.82 | 20.75 | 15.99 | 4.19 | 24.24 | 5.17 | 9.31 | 3.5 |

### 5.5.2 Discussion

Here, we tried to specify an $\alpha$ based on the ratio $\|G\|_F/\|c\|$, while there might be other properties of each instance—unknown to us at this point—that we could consider to systematically obtain an $\alpha$ that leads to considerable speedups.

## REFERENCES

M. S. Bazaraa, H. D. Sherali, and C. M. Shetty. *Nonlinear Programming: Theory and Algorithms*. Wiley-Interscience, third edition, May 2006. ISBN 9780471486008.

R. G. Bland. New finite pivoting rules for the Simplex method. *Mathematics of Operations Research*, 2(2):103–107, 1977. ISSN 0364765X.

Y. Chang and R. W. Cottle. Least-index resolution of degeneracy in quadratic programming. *Mathematical Programming*, 18:127–137, 1980. ISSN 0025-5610. DOI: 10.1007/BF01588308. 10.1007/BF01588308.

G. B. Dantzig. *Linear programming and extensions*. Princeton University Press, New Jersey, 1963.

G. B. Dantzig, A. Orden, and P. Wolfe. The generalized simplex method for minimizing a linear form under linear inequality restraints. *Pacific Journal of Mathematics*, 5:183–195, 1955.

R. Fletcher. Degeneracy in the presence of roundoff errors. *Linear Algebra and its Applications*, 106(0):149–183, 1988. ISSN 0024-3795. DOI: 10.1016/0024-3795(88)90026-2.

R. Fletcher. Resolving degeneracy in quadratic programming. *Annals of Operations Research*, 46-47:307–334, 1993. ISSN 0254-5330. DOI: 10.1007/BF02023102.

J. J. Forrest and D. Goldfarb. Steepest-edge Simplex algorithms for linear programming. *Mathematical Programming*, 57:341–374, 1992. ISSN 0025-5610. DOI: 10.1007/BF01581089.

D. Goldfarb and J. K. Reid. A practicable steepest-edge Simplex algorithm. *Mathematical Programming*, 12:361–371, 1977. ISSN 0025-5610. DOI: 10.1007/BF01593804.

N. I. M. Gould, D. Orban, and P. Toint. CUTEr and SifDec: A constrained and unconstrained testing environment, revisited. *ACM Transactions on Mathematical Software*, 29:373–394, December 2003. ISSN 0098-3500. DOI: 10.1145/962437.962439.

N. I. M. Gould, D. Orban, and D. P. Robinson. Trajectory-following methods for large-scale degenerate convex quadratic programming. Cahier du GERAD G-2011-50, GERAD, Montréal, Québec, Canada, 2011. To appear in Mathematical Programming Computation.

P. M. J. Harris. Pivot selection methods of the Devex LP code. In R. W. Cottle, L. C. W. Dixon, B. Korte, M. J. Todd, E. L. Allgower, W. H. Cunningham, J. E. Dennis, B. C. Eaves, R. Fletcher, D. Goldfarb, J.-B. Hiriart-Urruty, M. Iri, R. G. Jeroslow, D. S. Johnson, C. Lemarechal, L. Lovasz, L. McLinden, M. J. D. Powell, W. R. Pulleyblank, A. H. G. Rinnooy Kan, K. Ritter, R. W. H. Sargent, D. F. Shanno, L. E. Trotter, H. Tuy, R. J. B. Wets, E. M. L. Beale, G. B. Dantzig, L. V. Kantorovich, T. C. Koopmans, A. W. Tucker, P. Wolfe, M. L. Balinski, and Eli Hellerman, editors, *Computational Practice in Mathematical Programming*, volume 4 of *Mathematical Programming Studies*, pages 30–57. Springer Berlin Heidelberg, 1975. ISBN 978-3-642-00766-8. DOI: 10.1007/BFb0120710.

I. Maros and C. Meszaros. A repository of convex quadratic programming problems. *Optimization Methods & software*, 11-2(1-4, SI):671–681, 1999. ISSN 1055-6788. DOI: 10.1080/10556789908805768.

V. Raymond, F. Soumis, A. Metrane, and J. Desrosiers. Positive edge: A pricing criterion for the identification of non-degenerate Simplex pivots. Cahier du GERAD G-2010-61, GERAD, Montréal, Québec, Canada, 2010.

M. Towhidi and D. Orban. Customizing the Solution Process of COIN-OR's Linear Solvers with Python. Cahier du GERAD G-2012-07, GERAD, Montréal, Québec, Canada, 2012.

P. Wolfe. The Simplex method for quadratic programming. *Econometrica*, 27(3):382–398, 1959.

P. Wolfe and L. Cutler. Experiments in linear programming. In Graves and Wolfe, editors, *Recent Advances in Mathematical Programming*. McGraw-Hill, New York, 1963.

# CHAPTER 6

# GENERAL DISCUSSION

We prepared the way to investigate the possibility of utilizing LP degeneracy techniques on degenerate QPs. To this end, we took three steps.

In Chapter 3, we demonstrated the full potential of the positive edge pivot rule for the Simplex method by providing an efficient implementation.

To incorporate positive edge into a QP method we chose Wolfe's QP approach. Being a Simplex-like method, it requires a modification to the primal Simplex pivot rule. The recurrent need to implement and test pivot rules motivated us to create a framework in which these rules can be defined easily in a high-level programming language and passed to a solver, which is often written in a low-level programming environment. This resulted in the creation of open-source software called CyLP. In addition to its original purpose, CyLP enables customization of the solution process in MIPs, by defining specialized cuts and tree traversal strategy in branch and cut techniques, using Python. It is also equipped with a modeling tool that simplifies the definition of LPs and MIPs.

Wolfe's method proved to be inefficient on QPs, due to the large number of constraints it considers to solve a problem. On top of this, Wolfe's approach contains a procedure that accepts or rejects a pivot. In the tests, Wolfe's method usually rejected compatible pivots proposed by positive edge and thus nullified its effectiveness. On a second try, we considered reduced gradient methods, of which the Simplex method can be considered a special case. It outperforms Wolfe's method on general QPs by a remarkable margin. Combined with the positive edge rule, it demonstrates an inconsistent enhancement of performance on general test problems, making it difficult to draw a general conclusion about its behavior. However, if we add relatively small quadratic terms to the degenerate LPs—those on which we obtain noticeable speedups using positive edge— the reduced gradient method combined with positive edge is reporting faster execution times. The speedup diminishes as the norm of the quadratic part increases, motivating us to use scaling to alter the relative strength of the quadratic part against the linear section.

Scaling reduces the execution time of the reduced gradient method as well as the reduced gradient equipped with positive edge. However, we were unable to provide a systematic way

of determining the scale factor, based on the ratio of the Hessian norm to the norm of the linear coefficients.

In this research, we considered incorporating the postive edge method into QP solution techniques, Wolfe's method and the reduced gradient method. These methods were our first choice because of their similarity to the Simplex method. Another approach would have been to choose the method of Goldfarb [1986], which is an efficient primal active-set method, and build a positive-edge-like technique on it that is able to take advantage of the problem's degeneracy.

# CHAPTER 7

# CONCLUSION

In Chapter 3, we described the development of an efficient implementation of the positive edge pivot rule for the Simplex method. Positive edge improves the well-known Devex pivot method significantly, reducing the solution time by a maximum factor of 4.23 on instances with 75% degeneracy. We also observed that, on average, it does not cause slow-downs on problems with less than 25% degeneracy.

In this thesis, we developed open-source software, called CyLP, that is built upon CLP and allows definition of pivot rules in Python. It includes a modeling tool that, we believe, is easier to use than other available modeling options.

A natural continuation of this thesis would be to search for a conclusive result on the use of LP's degeneracy techniques on QPs. While we obtained good speedups with specific problems, we proved that techniques like positive edge are inefficient on particular problems. Therefore, a general statement about when and how positive edge should be applied on a QP remains to be found.

An interesting area of research is to investigate the application of the positive edge rule on methods that solve LPs iteratively in their process. One example of such methods is Sequential Linear Programming (SLP) for nonlinear programming (NLP) [Bazaraa et al., 2006]. At each step of this approach, the problem is linearized around the current point. The resulting LP is then solved to determine a step, subject to trust-region bounds. Updating the trust region based on the accuracy of the LP model guarantees convergence. If an NLP has linearly dependent constraints at its solution, it is likely that its linear model is also degenerate near the solution. Therefore, positive edge can be used to solve the linear model faster, reducing the iteration time and hence decreasing the total execution time.

Another example is the SLP-EQP method, proposed by Fletcher and de la Maza [1989] and improved by Byrd et al. [2003] to solve large-scale nonlinear programs. Consider a nonlinear problem with equality and inequality constraints. At each iteration, the step is computed in two stages. In the first stage, we solve an LP, the solution of which provides an estimate of the active set at the solution, or a working set. In stage two, using the working set, we solve a QP model of the problem, while we impose equality on the constraints in the working set

(EQP). Again, we can use positive edge to reduce the computation time of the first stage and, as a result, the total run time.

On the computational side, CyLP could be improved by allowing customization of the dual Simplex method. This is essential in the context of MIP, where we often use the dual Simplex method to solve the relaxation of the problem at each branch-and-cut node.

## REFERENCES

A. Aides. Cython wrapper for IPOPT. `http://code.google.com/p/cyipopt`. [Online; accessed 2-November-2011].

M. S. Bazaraa, H. D. Sherali, and C. M. Shetty. *Nonlinear Programming: Theory and Algorithms*. Wiley-Interscience, third edition, May 2006. ISBN 9780471486008.

M. Berkelaar. lpsolve, A Mixed Integer Linear Programming Software. `http://lpsolve.sourceforge.net`. [Online; accessed 2-November-2011].

R. G. Bland. New finite pivoting rules for the Simplex method. *Mathematics of Operations Research*, 2(2):103–107, 1977. ISSN 0364765X.

R. H. Byrd, N.I.M. Gould, J. Nocedal, and R. A. Waltz. An algorithm for nonlinear optimization using linear programming and equality constrained subproblems. *Mathematical Programming*, 100:27–48, 2003. ISSN 0025-5610. DOI: 10.1007/s10107-003-0485-4.

W. Carolan, J. Hill, J. Kennington, S. Niemi, and S. Wichmann. Empirical Evaluation of the KORBX Algorithms for Military Airlift Applications. *Operations Research*, 38:240–248, 1990.

Y. Chang and R. W. Cottle. Least-index resolution of degeneracy in quadratic programming. *Mathematical Programming*, 18:127–137, 1980. ISSN 0025-5610. DOI: 10.1007/BF01588308. 10.1007/BF01588308.

B. A. Cipra. The best of the 20th century: editors name top 10 algorithms. *SIAM News*, 33 (4):1–2, 2000.

CLP. COIN-OR Linear Programming. `https://projects.COIN-OR.org/Clp`. [Online; accessed 2-November-2011].

G. B. Dantzig. *Linear programming and extensions*. Princeton University Press, New Jersey, 1963.

G. B. Dantzig, A. Orden, and P. Wolfe. The generalized simplex method for minimizing a linear form under linear inequality restraints. *Pacific Journal of Mathematics*, 5:183–195, 1955.

E. Dolan and J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming B*, 91:201–213, 2002.

J. Dongarra and F. Sullivan. Guest editors' introduction: The top 10 algorithms. *Computing in Science and Engineering*, 2:22–23, 2000. ISSN 1521-9615. DOI: 10.1109/MCISE.2000.814652.

I. Elhallaoui, D. Villeneuve, F. Soumis, and G. Desaulniers. Dynamic aggregation of set-partitioning constraints in column generation. *Operations Research*, 53(4):632–645, 2005. DOI: 10.1287/opre.1050.0222.

I. Elhallaoui, G. Desaulniers, A. Metrane, and F. Soumis. Bi-dynamic constraint aggregation and subproblem reduction. *Computers and Operations Research*, 35(5):1713 – 1724, 2008. DOI: 10.1016/j.cor.2006.10.007.

I. Elhallaoui, A. Metrane, , G. Desaulniers, and F. Soumis. An improved primal simplex algorithm for degenerate linear programs. *INFORMS Journal on Computing*, 2010a. DOI: 10.1287/ijoc.1100.0425.

I. Elhallaoui, A. Metrane, F. Soumis, and G. Desaulniers. Multi-phase dynamic constraint aggregation for set partitioning type problems. *Mathematical Programming*, 123:345–370, 2010b. ISSN 0025-5610. DOI: 10.1007/s10107-008-0254-5.

R. Fletcher. A general quadratic programming algorithm. *Journal of the Institute for Mathematics Applications*, (7):76–91, 1971.

R. Fletcher. *Practical Methods of Optimization*. Wiley, second edition, 1987.

R. Fletcher. Degeneracy in the presence of roundoff errors. *Linear Algebra and its Applications*, 106(0):149–183, 1988. ISSN 0024-3795. DOI: 10.1016/0024-3795(88)90026-2.

R. Fletcher. Resolving degeneracy in quadratic programming. *Annals of Operations Research*, 46-47:307–334, 1993. ISSN 0254-5330. DOI: 10.1007/BF02023102.

R. Fletcher and E.S. de la Maza. Nonlinear programming and nonsmooth optimization by successive linear programming. *Mathematical Programming*, 43:235–256, 1989. ISSN 0025-5610. DOI: 10.1007/BF01582292.

J. J. Forrest and D. Goldfarb. Steepest-edge Simplex algorithms for linear programming. *Mathematical Programming*, 57:341–374, 1992. ISSN 0025-5610. DOI: 10.1007/BF01581089.

D. Goldfarb. Efficient primal algorithms for strictly convex quadratic programs. *Numerical Analysis*, 1230:11–25, 1986.

D. Goldfarb and A. Idnani. A numerically stable dual mehtod for solving stricly convex quadratic programs. *Mathematical Programming*, (27):1–33, 1983.

D. Goldfarb and J. K. Reid. A practicable steepest-edge Simplex algorithm. *Mathematical Programming*, 12:361–371, 1977. ISSN 0025-5610. DOI: 10.1007/BF01593804.

N. I. M. Gould and P. L. Toint. A quadratic programming bibliography, 2008.

N. I. M. Gould, D. Orban, and P. Toint. CUTEr and SifDec: A constrained and unconstrained testing environment, revisited. *ACM Transactions on Mathematical Software*, 29:373–394, December 2003. ISSN 0098-3500. DOI: 10.1145/962437.962439.

N. I. M. Gould, D. Orban, and D. P. Robinson. Trajectory-following methods for large-scale degenerate convex quadratic programming. Cahier du GERAD G-2011-50, GERAD, Montréal, Québec, Canada, 2011. To appear in Mathematical Programming Computation.

H. J. Greenberg. An analysis of degeneracy. *Naval Research Logistics Quarterly*, 33:635–655, 1986.

P. M. J. Harris. Pivot selection methods of the Devex LP code. In R. W. Cottle, L. C. W. Dixon, B. Korte, M. J. Todd, E. L. Allgower, W. H. Cunningham, J. E. Dennis, B. C. Eaves, R. Fletcher, D. Goldfarb, J.-B. Hiriart-Urruty, M. Iri, R. G. Jeroslow, D. S. Johnson, C. Lemarechal, L. Lovasz, L. McLinden, M. J. D. Powell, W. R. Pulleyblank, A. H. G. Rinnooy Kan, K. Ritter, R. W. H. Sargent, D. F. Shanno, L. E. Trotter, H. Tuy, R. J. B. Wets, E. M. L. Beale, G. B. Dantzig, L. V. Kantorovich, T. C. Koopmans, A. W. Tucker, P. Wolfe, M. L. Balinski, and Eli Hellerman, editors, *Computational Practice in Mathematical Programming*, volume 4 of *Mathematical Programming Studies*, pages 30–57. Springer Berlin Heidelberg, 1975. ISBN 978-3-642-00766-8. DOI: 10.1007/BFb0120710.

K. L. Hoffman and M. Padberg. Solving airline crew scheduling problems by branch-and-cut. *Management Science*, 39:675–682, June 1993. DOI: 10.1287/mnsc.39.6.657.

H. Koepke. Cython wrapper for CPLEX. `http://www.stat.washington.edu/~hoytak/code/pycpx/index.html`, a. [Online; accessed 2-November-2011].

H. Koepke. Cython wrapper for lpsolve. `http://www.stat.washington.edu/~hoytak/code/pylpsolve/index.html`, b. [Online; accessed 2-November-2011].

C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, 2nd Edition*. Prentice Hall, 2001.

R. Lougee-Heimer. The common optimization interface for operations research. *IBM Journal of Research and Development*, 47(1):57–66, 2003. DOI: 10.1147/rd.471.0057. URL `www.COIN-OR.org`.

A. Makhorin. GLPK, GNU Linear Programming Kit. `http://www.gnu.org/s/glpk`. [Online; accessed 2-November-2011].

I. Maros and C. Meszaros. A repository of convex quadratic programming problems. *Optimization Methods & software*, 11-2(1-4, SI):671–681, 1999. ISSN 1055-6788. DOI: 10.1080/10556789908805768.

J. J. Moré and G. Toraldo. Algorithms for bound constrained quadratic programming problems. *Numerical Mathematics*, 55:377–400, 1989.

J. J. Moré and G. Toraldo. On the solution of large quadratic programming problems with bound constraints. *SIAM Journal of Optimization*, 1(1):93–113, 1991.

J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, second edition, 1999.

M. Padberg and G. Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Rev.*, 33:60–100, February 1991. ISSN 0036-1445. DOI: 10.1137/1033004.

P.Q. Pan. A basis deficiency-allowing variation of the simplex method for linear programming. *Computers and Mathematics with Applications*, 36(3):33–35, 1998.

M.J.D. Powell. A fast algorithm for nonlinearly constrained optimization calculations. *Lecture Notes in Mathematics, Springer Verlag, Berlin*, (630):144–157, 1978.

PuLP. An LP modeler written in Python. http://code.google.com/p/pulp-or. [Online ; accessed 2-November-2011].

V. Raymond, F. Soumis, A. Metrane, and J. Desrosiers. Positive edge: A pricing criterion for the identification of non-degenerate Simplex pivots. Cahier du GERAD G-2010-61, GERAD, Montréal, Québec, Canada, 2010a.

V. Raymond, F. Soumis, and D. Orban. A new version of the improved primal simplex for degenerate linear programs. *Computers and Operations Research*, 37(1):91–98, 2010b.

P. J. S. Silva. Pycoin, interface to some COIN packages. http://www.ime.usp.br/~pjssilva/software.html, 2005. [Online ; accessed 2-November-2011].

T. Terlaky and S. Zhang. Pivot rules for linear programming: A survey on recent theoretical developments. *Annals of Operations Research*, 46-47:203–233, 1993. ISSN 0254-5330. DOI: 10.1007/BF02096264.

M. Towhidi and D. Orban. Customizing the Solution Process of COIN-OR's Linear Solvers with Python. Cahier du GERAD G-2012-07, GERAD, Montréal, Québec, Canada, 2012.

S. A. Vavasis. Quadratic programming is in NP. *Information Processing Letters*, 36(2):73–77, 1990.

S. A. Vavasis. *Nonlinear Optimization: Complexity Issues*. Oxford University Press, 1991.

A. Wächter and L. T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106:25–57, 2006. DOI: 10.1007/s10107-004-0559-y. URL https://projects.COIN-OR.org/Ipopt.

P. Wolfe. The Simplex method for quadratic programming. *Econometrica*, 27(3):382–398, 1959.

P. Wolfe and L. Cutler. Experiments in linear programming. In Graves and Wolfe, editors, *Recent Advances in Mathematical Programming*. McGraw-Hill, New York, 1963.